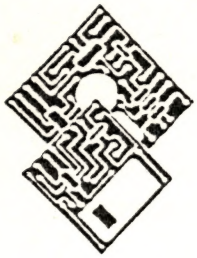
R 10/28/87



APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION

290 SW 43rd. Street
Renton, WA 98055
206-251-6548

APPLE IIGS BASIC

Version 1.0B4

APDA# K2SBAB

Apple® II Apple IIGS™ BASIC

APDA draft

September 11, 1987

Apple® Software Publications

This document contains preliminary information. It does not include:

- final editorial corrections
- final art work
- index

It may not include final technical changes.

APPLE COMPUTER, INC.

Copyright © 1987 by Apple Computer, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

Apple, the Apple logo, AppleTalk, ImageWriter, LaserWriter, and ProDOS are registered trademarks of Apple Computer, Inc.

Apple Desktop Bus, Apple IIGS, AppleWorks, Macintosh, and SANE are trademarks of Apple Computer, Inc.

ITC Avant Garde Gothic, ITC Garamond, and ITC Zapf Dingbats are registered trademarks of International Typeface Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

POSTSCRIPT is a trademark of Adobe Systems Incorporated.

Simultaneously published in the United States and Canada.

ISBN 0-201-xxxxx-x
ABCDEFGHIJ-DO-8987
First printing, Nnnn 1987

Chapter 1

Introduction to Apple IIGS BASIC

Starting up xx

Entering statements xx

Immediate and deferred execution xx

Editing while typing xx

Entering your program xx

Syntax checking xx

Editing your program xx

Interrupting a running program xx

Controlling text on the screen xx

The LIST statement xx
The reserved variable LISTTAB xx
The reserved variables INDENT and OUTREC xx
The DEL statement xx
The TEXTPORT statement xx
The TEXT statement xx
The reserved variables VPOS and HPOS xx
The HOME statement xx
The INVERSE and NORMAL statements xx

System and utility statements xx

Memory management xx
- The NEW command xx
- The CLEAR command xx
- The reserved variable FRE xx

Loading and saving programs xx
- The LOAD statement xx
- The SAVE statement xx

Starting and stopping programs xx
- The RUN statement xx
- The STOP statement xx
- The END statement xx
- The CONT statement xx

Handling large programs xx
- The CHAIN statement xx

Debugging programs xx
- The TRACE statement xx
- The NOTRACE statement xx

Special keyboard functions xx
- The Control-C combination xx
- The Control-Reset combination xx

Automatic execution xx
- The EXEC statement xx
  - Deferred immediate statements xx
  - Creating deferred statements xx
- Capturing programs as text files xx

Apple IIGS™ BASIC is an extended version of the BASIC programming language for the Apple IIGS computer. The following are some of its more important features:

- The facilities for using disk files are built in, so that programs can easily read and write files, including special, easy-to-use type-tagged Data file input and output of all types of BASIC variables.
- The double-precision real variable type provides arithmetic with 15 digits of precision and a numeric range of 10^{-308} to 10^{+308}.
- The long-integer variable type provides 19-digit precision, and the double-integer variable type provides 9 digits for address manipulation.
- The PRINT USING and IMAGE statements are powerful tools for controlling the exact format of data displayed or printed by a BASIC program.
- The INPUT USING and IMAGE statements are specialized tools for controlling the input text fields and lines by a BASIC program.
- Variable names may be up to 30 characters long. Reserved words are recognized only when they are set off either by spaces or characters other than letters, the period or digits. This allows you to use reserved words embedded in variable names without causing syntax problems.
- Any line in a program may have an optional label, and it can be referenced by the label or by the line number in a GOTO, GOSUB, or similar statement.
- A full-featured command line editor, with insert and replace modes, and an EDIT command for editing existing program lines one at a time are available.
- Complete support for access to the Apple IIGS Toolbox, via CALL, CALL%, and LIBRARY statements, as well as language extension with external assembly-languages modules, through INVOKE, EXFN, and PERFORM statements.
- A Window Manager and Menu Manager interface, through EVENTDEF, MENUDEF, and TASKPOLL statements, provides a simple and efficient link to Task Master for writing mouse-based desktop application programs in GS BASIC.
- A special form of the OPEN command can open a QuickDraw II GrafPort or a Window Manager window port for applying PRINT # and PRINT # USING statements directly to the graphics screen.

Starting up

Before you read any further, load Apple IIGS BASIC so that you can follow along with the examples in this manual.

1. Begin with the system plugged in and the power switch off.
2. Insert the Apple IIGS BASIC diskette into the system disk drive (use the method shown in your *Apple IIGS Owner's Guide).*

3. Turn the power switch on. The system will start up, and the drive's in-use light will come on for a few moments while the information on your BASIC diskette is being loaded into the Apple IIGS system memory.
4. Soon you will see a prompt line at the top of the screen, giving the version of Apple IIGS BASIC that you are running. Below this line is a right parenthesis character at the left margin of the video display screen, with a cursor to its right. Apple IIGS BASIC is up and running. (See your *Apple IIGS Owner's Guide* if you have difficulty starting up your computer.)

If there is a BASIC program file named GSB.HELLO on the diskette that you use when starting up your computer (as there is on the Apple IIGS BASIC diskette), that program will automatically be executed when the system starts. This feature lets you develop turnkey systems easily; simply name your working program GSB.HELLO.

The right parenthesis is BASIC's **prompt** character. When it appears on the screen, it means that BASIC is waiting for you to type something. The cursor, which appears as a vertical blinking rectangle alternating between the current text color and the background color, shows you where the next character that you type will appear on the screen. If you are using a color video monitor, you can change the colors for the text and the background (and the border) in the Control Panel. (See Appendix A in the *Apple IIGS Owner's Guide* for details.)

Entering statements

Now that you have the prompt character and the cursor on the screen, you are ready to begin using the Apple IIGS BASIC language. Before you do anything else, enter the command NEW and press the Return key. NEW tells your computer that you are entering a new program.

Then type exactly as shown here:

```
PRINT "Hi there"
```

and press Return. Your computer will display

```
Hi there
```

followed by the prompt and the blinking cursor.

This statement instructed your computer to display all of the characters between the quotation marks.

If you misspell the reserved word PRINT, like this:

```
PRITN "Hi there"
```

You will see this error message:

```
?SYNTAX ERROR
```

The word *syntax* refers to the rules for constructing statements in the BASIC language. Unfortunately, not all your errors be so apparent. You may enter a statement that is syntactically correct, but does not say quite what you intended. BASIC will execute the statement exactly as you typed it. For example, if you type

```
PRINT Hi there
```

The computer will display

```
0
```

Since there were no quotation marks, the computer displayed a zero, the value of the variable HITHERE.

When you finish typing a statement, press the Return key, and BASIC will try to understand what you typed and carry out your instructions.

Don't press the Return key if you are typing a long statement and getting close to the right margin of the display screen. When you reach the end of the first screen line, BASIC automatically continues the statement on the next screen line. BASIC will allow you to enter three screen lines of text before reaching its right margin.

The prompt character will not scroll to the bottom line of the screen; it always stays at least three lines above the bottom.

Immediate and deferred execution

The PRINT statement example was **executed** immediately after you pressed the Return key. This is called **immediate execution.** Now type

```
10 PRINT "Hi again"
```

and press Return. The number 10 is called a **line number.** Nothing appears on the screen except the BASIC prompt. Now type

```
RUN
```

and press the Return key. You should see this on the screen:

```
Hi again
)
```

BASIC has executed the PRINT statement. When you typed the PRINT statement with a number in front of it, BASIC stored the statement in memory as a one-statement program. When you typed the RUN statement, it was executed immediately because it didn't have a line number. RUN is a statement that tells BASIC to execute whatever program is currently stored in memory. Execution of statements previously stored in memory is called **deferred execution.**

Statements without line numbers are executed immediately and forgotten as soon as they are executed. Statements with line numbers are not executed immediately, but remain in your computer's memory and can be executed again and again with the RUN statement. Your computer executes the lines of instructions you type in numeric order, always beginning with the lowest number. For the time being, number your program lines by tens (10, 20, 30, and so on). You'll learn why later.

A deferred statement and the line number preceding it are often referred to as a program *line*. For example, consider

```
30 PRINT "Bye now!"
```

Line 30 comprises both the PRINT statement and the line number 30.

Let's add another line to the program. Type

```
20 PRINT "Hi yet again"
```

and press the Return key. Then type another RUN statement, and press Return. Once again your screen displays

```
RUN
Hi again
Hi yet again
)
```

You can see that the statement at line number 10 is executed before the statement at line number 20. BASIC always executes statements in a program in the order of their line numbers, unless the program statements tell it to alter that sequence. Note that the order in which statements are executed is not necessarily the order in which you typed them.

❖ *By the way:* Now that you have learned that every program line you enter must be ended by pressing the Return key, we won't bother to include that step in the remaining instructions.

Editing while typing

You can correct errors that you catch while entering immediate or deferred statements before you press Return (called command line editing) by using the editing keys described below.

Left (¨)and Right (Æ) Arrow keys	Move the cursor to the error within the text you have entered. You can retype right over the mistake.

Delete key	Delete unwanted characters by positioning the cursor just to the right of an incorrect letter and pressing this key. This will erase the characters to the left of the cursor. You may repeat, moving the cursor backwards through the text as you do so.
Control-D	Hold down the Control key and then press and release the D key to perform the same function as the Delete key.
Control-F	Hold down Control, then type F to delete the character *under* the cursor, rather than the one to its left. You may repeat this to delete forwards through the text; the cursor will remain stationary, and the text will shift to the left as characters are removed.
Control-X	Hold down Control, then press X to erase the entire program line. This will erase everything you have typed since the last time you pressed Return. You can now retype the entire program line (a program line can actually be up to three lines on the screen).
Control-Y	Hold down Control, then press Y to erase all the characters from the current cursor position to the end of the text. To use Control-Y, first position the cursor on the first character you want deleted, and then press Control-Y. The character under the cursor and everything to its right will be erased, but characters to the left of the cursor will remain unchanged.

Let's experiment with these editing features now. First, just hold down a letter key until three lines fill up with the same letter. Note that **auto-repeat** will speed up the entry of these letters, following a short pause after the first letter. When you approach the end of the third line, release the key, and then press it repeatedly to display one character at a time to the end.

When the cursor is in the last position of the third line, the computer will beep if you type another character. You have now reached the right margin. Now try backing up with the Left Arrow key (←) to somewhere in the middle of the second line and press Control-Y. All the letters from the cursor to the end will be erased.

Now press Control-X to erase all the remaining characters.

❖ *Note:* Applesoft BASIC programmers should be aware that all the characters you have typed, even those to the right of the cursor, are considered part of your input line after you press Return.

Up to this point, our command line editing has been in **replace mode,** which is indicated by the blinking block cursor, the **replace cursor.** In replace mode, whenever you type a character, it replaces the character under the cursor.

Another mode, called **insert mode,** allows you to insert new characters between ones you have already typed. This mode is indicated on the screen by the **insert cursor.** To see the insert cursor, press Control-E. The cursor will become a blinking underline character. Now press Control-E again, and the replace cursor will reappear. The Control-E edit key functions as the cursor **toggle switch.**

Let's see how the different editing cursors work.. Type the line

```
Try out the old cursor
```

Move the cursor over the beginning of the word *old* and type in *new.*

```
Try out the new cursor
```

The word *new* is typed right over the previous word.

Now change to the insert cursor by pressing Control-E. You will see the blinking underline character. Move to the space between *the* and *new* by pressing the Left Arrow key and type *snazzy.*

```
Try out the snazzy new cursor
```

The new word is inserted into the text. Existing characters are shifted to the right as new characters are typed.

Press Control-X to erase the text before you press Return.

Entering your program

What if you make a mistake and don't catch it before you press Return? If it is a statement for immediate execution, you may get an error message right away. If you make a spelling or grammar error in typing a statement with a line number, BASIC may not detect the error until you type RUN and it tries to execute the statement. For example, if you enter the statement

```
15 PRINT INPUT "HELLO"
```

into your program, BASIC won't find the syntax error until it attempts to execute line 15. At that point, execution stops, and BASIC displays the message

```
?SYNTAX ERROR IN 15
```

where 15 is the line number of the incorrect statement. You should correct the erroneous statement before you run the program again.

You can make changes to existing program lines in two ways. The first method is to simply type a new line with the same line number. For example, if you type

```
15 PRINT "Hi another time"
```

this new version of line 15 replaces the old one. If you want to delete a stored statement altogether, type just its line number, then press Return. The second way is to use the EDIT command as described later in this chapter.

It's a good idea to leave room for additional line numbers between the numbers you use when you write a program. That way, if you want to insert a new program line between two lines already in memory, you can give it one of the unused numbers. For example, if you have entered two lines numbered 15 and 20, you can put a new line between them by giving it the number 18. Line numbers must be within the range of 1 to 65279, or a

```
?ILLEGAL LINE#/LABEL ERROR
```

message will be displayed on the screen.

The AUTO command provides an easy way to enter lines in your program and have BASIC automatically generate the line numbers for you. To use the AUTO command, simply type

```
AUTO linenum
```

You must select a linenum as the starting line number for the AUTO command. When you enter the **auto-edit mode,** the screen is split into two parts. The line number you requested will appear in a four-line entry window at the bottom of the screen, followed by the cursor. You can now type a statement and then press Return.

The statement will be checked and displayed in the upper 20 lines of the screen. Then the next line number will be displayed in the entry window. To exit from auto-edit mode, press the Esc (Escape) key, and the screen split will disappear. Other features of the AUTO command are described in Chapter 8, "BASIC Reference."

If you want to see your program as currently stored in memory, type

```
LIST
```

All statements in memory will be displayed on your screen in order of their line numbers.

If you want to get rid of all stored statements to start a new program, type

```
NEW
```

Be sure that you don't want to save the program in memory before you type NEW, since there is no way to recover the program after you press Return, unless you first save the program on a diskette. If you type LIST after you type NEW, you will see that the program previously stored in memory is gone.

Syntax checking

BASIC does limited syntax checking when statements are entered in direct mode. When error messages are displayed for direct mode entry (but not during an EXEC), two lines are displayed on the screen below the text you entered. The first line is mostly blank, except for the pointer character, the caret (^). BASIC displays one line with the caret at the position where it failed when scanning the statement.

Usually, but not always, the caret will point at the cause of the error message or at the end of a reserved word that was used incorrectly. The process of scanning your statement is called **tokenization.** This refers to the fact that all the BASIC reserved words are converted into tokens which only require 1 or 2 bytes instead of 1 byte for every character in the reserved word (this makes the program smaller that its equivalent text).

BASIC does a limited check of the syntax of each statement in a line, but does not verify the entire syntax of each statement that you enter. The syntax of a program line is described as either of the following:

linnum [*label:] statement* return
linnum [*label:*]*statement* [{: *statement }]* return

The bracket characters are used to indicate that the element they enclose is optional; you should not type the bracket or brace characters when entering a line. The simplest case, shown first, has only one statement in the line. This first description means that a deferred line consists of a line number optionally followed by a label ended with a colon, then a BASIC statement followed by a Return.

The brace characters used in the second definition indicate that the elements they enclose may be repeated. The second description adds the concept that a program line may have multiple statements separated by colons; the brackets indicate that the colon and the second statement are optional, and the braces indicate that the colon-statement sequence may be repeated as many times as necessary.

The general syntax of a statement is described as:

verb [{*nouns adverbs numbers characters*}]
or *variable*= {*nouns adverbs numbers characters*}
or *variable*({ *nouns adverbs numbers characters*}

The vertical-bar characters are used to indicate that the elements they separate are alternatives; you should not type the vertical-bar, bracket, or brace chararcters when entering a line. In other words, a statement is either one of the following:

- a verb optionally followed by one or more nouns, adverbs, numbers or any other valid characters (such as arithmetic operators) in the proper sequences.
- a variable name followed by either an equal sign or a left parenthsis and some collection of other elements that make up an implied LET statement.

For the purposes of this description, nouns and adverbs are specific subsets of the list of reserved words known to BASIC. Generally, a BASIC noun is a reserved word that has a value, or returns a value. For example the reserved word SIN is a function that returns a value but SIN is never a valid verb or a variable name and cannot be used to begin a statement.

BASIC adverbs are reserved words used to separate clauses within the syntax of a statement begun with a verb. For example, the statement

```
FOR I = 1 TO 20 STEP 3
```

contains the verb FOR and the two adverbs TO and STEP. Adverbs are like nouns in that they never begin a statement. Some verbs, like FOR, are also used as adverbs by some other verb, as in

```
OPEN ... FOR UPDATE
```

The BASIC tokenizer requires that statements begin with a verb or a variable name followed by an equal sign or a left parenthesis. If you begin a statement with a noun or an adverb BASIC will display the caret pointing at the last character of the invalid word, followed by the message

```
?RESERVED WORD ERROR
```

on the next line. If you use a verb that can only begin a statement as an adverb, this same message will appear. The specific reserved words in each subset considered here are described in Appendix C, "Reserved Words."

Starting and stopping programs

You may want to stop a program while it is running or restart a stopped program. The statements that allow you to stop and restart programs are described below.

The RUN statement

RUN is used to start running a program. When you enter a RUN statement, BASIC clears all variables, closes all open files (except executing text files), and begins to execute the program in memory, beginning with its smallest line number.

If there is no program in memory, the cursor drops to the next screen line, and BASIC redisplays the prompt. For example, to execute whatever program is currently in memory, type

```
RUN
```

You can specify program execution to begin at a line other than the smallest line number by following the command with that number. For example, to begin execution at line 205, type

```
RUN 205
```

If you specify a nonexistent line number, the

```
?UNDEF'D STATEMENT ERROR
```

message appears.

If you want to run a program that is in a disk file rather than stored in memory, you can specify the program to be run by giving its pathname. For example, to run the program stored in a file named ASSETS, use

```
RUN ASSETS
```

And if you wanted to begin execution at line 7254 of that program, use

```
RUN ASSETS, 7254
```

If BASIC cannot find the file that you specified after searching the disk, it displays the

```
?FILE NOT FOUND ERROR
```

message. If the file is found, the current program and all of its variables are erased from memory, and all open files (except an executing text file) are closed. The program specified in the RUN statement is then stored in memory, and BASIC begins executing it at either the lowest numbered line or at the specified line.

The RUN statement is the last statement executed in any line. For example, in the line

```
IF X=1 THEN RUN 1000 : I=1
```

the variable I will never be assigned the value 1.

Trying to run a program that is not written in BASIC generates a

```
?FILE TYPE ERROR
```

message.

The STOP statement

IF BASIC encounters the STOP statement while a program is running, it halts execution of the program, closes any executing text file, returns BASIC to immediate execution, resets the output file to the console, redisplays the prompt, and displays a message. For example, the message

```
?PROGRAM INTERRUPTED IN 8712
```

will appear if 8712 is the number of the program line containing the STOP statement. The program in memory is not altered in any way. STOP has no options associated with it.

The END statement

END is the same as STOP, except that no message is displayed when it is executed. The END statement has two options used in conjunction with the DEF statement. See Chapter 7, "Advanced Topics," and Chapter 8 "BASIC Reference," for more details.

The CONT statement

CONT causes execution of a program that has been halted by a STOP or END statement, or by a Control-C, to resume. The CONT command resumes execution at the statement immediately following the one at which execution was suspended, not with the first statement in the next program line. For example:

```
)5 PRINT"Program begins"?
)10 END: PRINT "Back again"?
)20 PRINT "All Done"
)RUN
Program begins
)CONT
Back again
All Done
)
```

CONT does not clear the program or reset the variables in memory, and there are no options associated with it. CONT has no effect if there is no program in memory.

You can continue a program halted by an error by using the CONT command. BASIC will attempt to continue execution starting with the statement in which the error occurred. An error made in immediate execution will not prevent a program from being continued.

A program that has had any of its statements altered, or any new statements added, cannot be continued with the CONT command. If you try, you will see the message

```
?CAN'T CONTINUE ERROR
```

Variables in a program can be changed using assignment statements in immediate execution. For example:

```
)10 X=4 : PRINT X
)20 STOP
)30 PRINT X
)RUN
4

BREAK IN 20
)X=X+2
)CONT
6
)
```

The CONT command can only be used in immediate excecution mode, and it will display an error message if you use it within a program.

Handling large programs

Even with the large amount of memory available for Apple IIGS BASIC programs, you may find ways to fill it up. Then too, it is usually not a good idea to pack thousands of program lines into a single program, since the result is hard to read and difficult to modify. The flexibility to divide large programs into easier to handle pieces is provided by the CHAIN statement, described below.

The CHAIN statement

If a program requires more memory than is available on your computer, or you want to divide it up logically, you can split the program into smaller pieces and execute them individually with the CHAIN statement.

CHAIN automatically loads and runs a specified program, without clearing the values of the variables left over from the previous program or closing any files the previous program left open. This allows variables used in one program to be used in another. The pathname of the program to chain must follow the reserved word CHAIN. For example, to chain to a program named Tires, use

```
CHAIN Tires
```

If the program specified in the CHAIN statement does not exist on the diskette, then a

```
?FILE NOT FOUND ERROR
```

message will be displayed.

Execution of the specified program begins at the smallest line number, unless you specify otherwise. Therefore

```
CHAIN /Link/Fence, 800
```

causes execution to begin at line 800 of the program named Fence on the volume /Link.

If the chained program uses a variable that was not used in the program executed before it, a new variable will be created; otherwise, the old variable will be used.

If the chained program dimensions an array that was dimensioned in the previous program, a

```
?DUPLICATE DEFINITION ERROR
```

message appears.

Here is an example of how you might use the CHAIN statement:

```
)NEW
)10 PRINT "Program Two speaking"
)20 PRINT "A(32) is ";A(32)
)30 A(32) = 42
)40 CHAIN ProgramOne, 70 :REM Go back to Program One
)Save ProgramTwo
)NEW
)10 PRINT "Program One"
)20 DIM A(44)
)30 FOR i=1 TO 44
)40 A(i) = i
)50    NEXT i
)60 CHAIN ProgramTwo : REM Now Program Two will be run
)70 PRINT "Back to Program One" : REM Back from Program Two
)80 PRINT "A(32) is now ";A(32)
)90 END
)Save ProgramOne
```

Running this program displays:

```
)RUN
Program One
Program Two speaking
A(32) is 32
Back to Program One
A(32) is now 42
)
```

Notice that you may chain back and forth between separate programs, and variable values are preserved throughout.

Debugging programs

Although Apple IIGS BASIC can easily detect errors such as unknown verbs and faulty syntax, you will have to find the more subtle sorts of errors such as slightly misspelled variable names. The TRACE statement can help you to catch these types of errors.

The TRACE statement

TRACE functions while a program is executing. It prints a number sign (#) followed by the number of each line of the program as it executes. It is very useful when tracing parts of the program that do not follow in sequential order.

TRACE used without any options displays the line numbers on the screen. If the program that you are running displays characters on the screen, the output of TRACE combines with that display in unpredictable ways. The line numbers may appear around and within, or even be overlayed by the program's display.

Fortunately, the TRACE statement has an option for directing its output to a file or another device, such as a printer. The syntax for this is:

```
TRACE TO #filenum
```

This option allows you to open a file or a device with the open command, and then send all the trace information to a disk, RAM disk file, or printer. You could even send the trace information through a serial connection to a separate terminal or computer and display the trace information on another screen.

If you trace a program that uses the OUTPUT# statement, the line numbers listed by TRACE will be included in the file written to by OUTPUT#.

TRACE is canceled by NOTRACE, RUN NEW followed by a pathname, or by LOAD followed by a pathname. TRACE is not canceled by CHAIN or RUN alone.

Warning:

Using TRACE with a program that includes an ON KBD statement can be risky because TRACE slows program execution, and many keys could be pressed while the ON KBD statement is being executed. This might cause a stack overflow error, giving a false error message for the program.

The NOTRACE statement

NOTRACE simply cancels TRACE, stopping the display of the line numbers of executing program statements. To use it, type

```
NOTRACE
```

There are no options associated with NOTRACE.

Special keyboard functions

Control-C and Control-Reset have special functions for users of Apple IIGS BASIC, as described below. Refer to your *Apple IIGS Owner's Guide* for information about other key combinations that have special functions.

The Control-C combination

Pressing Control-C while a statement in a program is being executed is equivalent to inserting a STOP statement immediately after the statement. Control-C can be used to stop the execution of any statement. For example, you could use it to terminate the display generated by a LIST statement.

Pressing Control-C while a program is waiting at an INPUT statement, before the Return key is pressed, will abort that program.

Control-C will not stop execution of a program in the following cases:

- A BREAK OFF or ON BREAK statement has been executed (Control-C is handled by ON BREAK; see Chapter 8, "BASIC Reference," for details).
- An ON KBD statement has been executed (ON KBD causes Control-C to be treated like any other keystroke). However, the statement executed by ON KBD can issue an END or STOP command if Control-C is pressed.

- The program is waiting for an input/output (I/O) operation to be completed. For example, if the printer is not properly connected while the program is trying to print, Apple IIGS BASIC will not recognize Control-C. The printer connection must be adjusted before the program can continue or be aborted by Control-C.

The Control-Reset combination

Pressing the Reset button while holding down the Control key halts I/O operations and reboots your computer. Control-Reset causes an electrical reset of the entire system. Anything stored in memory is lost after pressing Control-Reset, including your program and Apple IIGS BASIC.

Automatic execution

In addition to immediate execution (directed from the keyboard) and deferred execution (directed by programs stored in memory and started by a RUN command), Apple IIGS BASIC allows operation to be directed from a text file. You use the EXEC statement for automatic execution of instructions.

The EXEC statement

EXEC simulates keyboard input by reading the contents of a text file and executing its instructions. To use this statement, enter the reserved word EXEC, followed by the pathname of the text file containing the commands to be executed. For example:

```
EXEC  /Workdisk/Business/Gamestarter
```

If the file you have specified is not a text file, the

```
?FILE TYPE ERROR
```

message is displayed.

Apple IIGS BASIC accepts input only from the file specified in the EXEC statement until one of the following occurs:

- Control-C is pressed
- a STOP statement is executed
- an error message is displayed
- the end of the file is reached

After any of the above events, BASIC returns control to the keyboard.

EXEC automatically opens the file that it uses, but BASIC does not consider this file as one of the total of six files that may be open at one time.

Deferred immediate statements

You may have an application that is actually a series of programs, running one after the other. If the individual programs require no user interaction, you can use EXEC to run the sequence of programs, and then leave the operation unattended while the computer works. You can do this by creating a text file that contains the RUN statements, and then issuing an EXEC statement specifying that file. For example, you would type

```
EXEC Runner
```

where Runner is a text file containing the lines

```
RUN Program1
RUN Program2
RUN Program3
```

You can write programs that produce Files to be run by the EXEC statement. The following program, called TextfileMaker, is an example of such a program:

```
10 REM Program "TextfileMaker"
20 OPEN "Runner",FILTYP= TXT FOR OUTPUT AS #3
30 INPUT "-"; S$ : REM Get a line of text
40 IF LEN(S$)=0 THEN 70 : REM Was only Return pressed?
50 PRINT #3; S$ : REM Write the line into the file
60 GOTO 30 : REM Go back and get another line
70 CLOSE #3
80 END
```

TextfileMaker displays a hyphen to prompt you to enter lines of text, one at a time, until you type nothing but a Return. The lines will be written into a text file named Runner. If you run TextfileMaker and enter the following responses:

```
RUN
-RUN ProgA
-RUN ProgB
-RUN ProgC
-
```

Apple IIGS BASIC will create the desired EXEC file. When you type

```
EXEC Runner
```

the contents of the Runner file will be output, one line at a time, exactly as though you were entering the data from the keyboard. When the line containing RUN ProgA is output, BASIC executes that line as a RUN statement, and runs ProgA. When ProgA is finished running, the next line of text is output, causing ProgB to run. After ProgB finishes, ProgC is RUN. After ProgC finishes, control returns to the keyboard.

Note that you would not get the same result by writing a program to run the three programs. A program containing the lines

```
10 RUN "ProgA"
20 RUN "ProgB"
30 RUN "ProgC"
```

runs only the first program because executing the RUN statement clears the current program in memory, thus wiping out lines 20 and 30.

EXEC accepts any legal statement in BASIC, including conditional statements. This allows you to make the order of program execution dependent on the result of a program's execution. For example, you might want to run ProgB after ProgA only if the value of X as determined by ProgA is negative. If X is positive, you might want to run ProgC instead. To do this, your Runner file should contain the following:

```
RUN ProgA
IF X<0 THEN RUN ProgB : ELSE RUN ProgC
```

Remember that an executing text file replaces keyboard input. If an INPUT or GET statement occurs in a program, it takes its input from the next line of the text file, *not* from the keyboard.

If you call an EXEC file from a program, the EXEC statement must be followed by END, as shown below:

```
10 A$="WoodFile"
20 EXEC A$ : END
25 INPUT A$
30 GOTO 10
```

You can reenter the calling program after the EXEC file is finished by using either RUN with a line number or GOTO with a line number as the last line of the EXEC file.

You can force the computer to take input from the keyboard while a text file is executing by opening the file .CONSOLE in your program or in the text file and doing file input (see Chapter 5 "File Handling").

Creating deferred statements

When a deferred statement (one with a line number) occurs in an executing text file, the effect is just as if you typed it on the keyboard. The line is stored in memory, and it can be run or saved.

Suppose that you have a program in memory (either one that you have just typed or one previously saved) and you execute a text file containing deferred statements. If these statements have line numbers different from those already in the program, the effect is to add the new statements to the program. If any new line has the same number as a line in the program, the new line replaces the old one.

Suppose that you write a set of programs using modulo functions defined for real variables. A modulo function takes the first expression and returns the remainder after dividing the first expression by the second expression, called the **modulus.** (The value returned by 7 modulo 5 is 2. 8 modulo 2 returns 0.) This is the same as the MOD function in Apple IIGS BASIC.

If we call the real variable A and the modulus B, we can define a function named ModB as follows:

```
30 DEF FN ModB(A)=(A/B - INT(A/B))*B
```

where B is replaced by an actual number. The result is meaningful only if A has a positive value.

If you wanted to have Mod12, Mod15, and Mod255 functions, you could begin each program with the lines

```
1 DEF FN Mod12(A)=(A/12 - INT(A/12))*12
2 DEF FN Mod15(A)=(A/15 - INT(A/15))*15
3 DEF FN Mod255(A)=(A/255 - INT(A/255))*255
```

Instead of typing these lines into each program that uses these functions, you can create a text file called Mods that contains them, and then enter

```
EXEC Mods
```

A program like TextfileMaker (shown in the previous section) could be used to create the Mods file. Before running TextfileMaker, alter line 20 to produce a file named Mods instead of Runner. When prompted, enter

```
RUN
-1 DEF FN Mod12(A)=(A/12 - INT(A/12))*12
-2 DEF FN Mod15(A)=(A/15 - INT(A/15))*15
-3 DEF FN Mod255(A)=(A/255 - INT(A/255))*255
-
)
```

Now whenever you want to write a program using these MOD functions, start by typing EXEC Mods (enter this command before running the program, but after loading it).

The same principle can be used whenever you have some lines to insert in several programs. Make sure that the line numbers in the text file and the line numbers in your program don't overlap.

Capturing programs as text files

Some of your programs may contain lines that you would like to insert in other programs. To do this, you must convert the lines into a text file, then insert them into other programs by using EXEC as described above. Use the OUTPUT# statement to send console output to a text file instead of to the video screen. If you then use LIST, all the output is written to the text file.

The resulting text file may be edited with any Apple IIGS editor that accepts an ASCII file. After editing and saving the text file, you can use EXEC to read the edited program back into memory, and save the program from memory into a program file.

Suppose that you want to save lines 20 through 150 of a program starting at line 10 in a text file name Goodlines. First load the program, then type the following:

```
)OPEN "Goodlines", FILTYP=TXT FOR OUTPUT AS #3
)OUTREC=0 : INDENT =0
)OUTPUT #3 : LIST 20 - 150 : OUTPUT #0
)OUTREC=80 : INDENT =2
)CLOSE #3
```

This creates a text file that you can insert into other programs with the command

```
EXEC Goodlines
```

Chapter 2

Tools of Your Trade

Variable types xx

Integer variables xx
Double-Integer variables xx
Reals xx
Strings xx

Reserved words and variables xx

Arrays xx

The DIM statement xx

Statements xx

Expressions xx

Arithmetic Expressions xx
    Arithmetic operator precedence xx
Logical expressions xx
    Logical operator precedence xx

Functions xx

String Functions xx
    The LEN function xx
    The STR$ function xx
    The CONV$ function xx
    The VAL function xx
    The CHR$ function xx
    The ASC function xx
    The HEX$ function xx
    The TEN function xx

The ERRTXT$ function xx
The SPACE$ function xx
The REP$ function xx
The PFX$ function xx
The UCASE$ function xx
The LEFT$ function xx
The RIGHT$ function xx
The MID$ function xx
The INSTR function xx
The SUB$ function xx
Numeric functions xx
The SIN function xx
The COS functions xx
The TAN function xx
The ATN function xx
The INT function xx
The RND function xx
The SGN function xx
The ABS function xx
The SQR function xx
The EXP, EXP1, and EXP2 functions xx
The FIX function xx
The LOG, LOG1, LOG2, and LOGB% functions xx
The NEGATE function xx
The ROUND function xx
The SCALB function xx
The CONV& function xx
The CONV function xx
The CONV$ function xx
The CONV% function xx
Miscelaneous functions xx
The BTN function xx
The FILE function xx
The JOYX and JOYY variable xx
The PDL function xx
The PEEK function xx
Defining your own functions xx
The DEF FN statement xx
Using a defined function xx
Remarks about defining functions

This chapter describes the tools that Apple IIGS BASIC provides for effective information handling. Here are some terms that you should know before continuing:

- A **variable** is a container to store a value. It can be thought of as a box that can hold a single value. Variables have limits on the types of things they can contain (just as a box can contain only things of a certain size). Also, there are certain tasks for which some variables are better suited than others; for example, a filing cabinet is not a suitable place to store fruit, but a wooden crate may be.
- Variables are referred to by their **names.** The box in the example above might be labeled junk; the fruit crate could be labeled apples2. Both junk and apples2 are legal variable names. A variable name is a sequence of characters beginning with a letter and followed by from 0 to 29 additional letters, digits, or periods. Lowercase letters in variable names are considered equivalent to their uppercase counterparts. For example, the names **junk** and **JUNK** refer to the same variable.
- Not all possible names can be used. Some **reserved** words are used by IIGS BASIC to refer to the language's statements and functions.
- A **constant** is an unchanging, or fixed, value. There are two kinds of constants in BASIC: numeric and string. A numeric constant is a value written as a number; a string constant is any sequence of characters enclosed in quotation marks. For example, 3.14159265 is a numeric constant; "dangle" and "463" are string constants. The numeric constant 463 and the string constant "463" do not represent the same value.

Variable types

There are six **variable types** in Apple IIGS BASIC: single-precision reals, (generally referred to simply as **reals**), double-precision reals (called double), integers, double integers, long integers, and strings. The first five types represent numbers of various kinds, the last type represents sequences of characters.

The type of a variable is determined by the last character of its name: $ for string, # for double precision, % for integer, @ for double integer, and & for long integer. In the absence of any of these special trailing characters, the variable type is considered to be real (single precision) by default.

Here are examples of names of the six variable types:

Table 2-1

Name	Type
`Myname$`	string
`Length`	real
`Bignum#`	double precision
`Marbles7%`	integer

`GPaddress@`	double integer
`Light.Years&`	long integer

Simple variables are created when they are first used in a program. When BASIC sees a variable name in a statement, it first checks to see if it already has a variable with that name. If it finds the name, it knows where in memory to find the value stored in the variable.

If BASIC doesn't find a simple variable matching the name already in memory, it immediately creates the new variable. It places the new variable name in the directory of variable names so it can be found later, then finds free space in memory to store the value that the variable will contain. It also notes the type of the new variable.

For numeric variables, BASIC stores the value 0 in the variable; for string variables, it stores an empty, or null, string in the variable.

Integer variables

Generally, an **integer** is any positive or negative whole number without a decimal point. IIGS BASIC supports three sizes of integer variables and converts integer constants in a program into four internal binary formats when an integer constant is found in a program statement. The numbers 3, -3, and 20,000,167 are examples of integer constants.

The first, or smallest size, integers are usually referred to as *integers without any qualification;* you should think of them as *word integers.*

An integer variable name must end with a percent sign (%). For instance, I% is the name of an integer variable. Integer variables can store values up to five digits long, from -32768 to 32767. Attempting to assign a value beyond this range to an integer variable generates the message

```
?OVERFLOW ERROR
```

Integers are displayed without leading zeros (with a leading minus sign if negative), followed by up to five digits without a decimal point.

Integers are useful when fractional parts of numbers are not needed. They can also be used to speed up some types of calculations where real numbers are not required, and they take up significantly less space in memory.

Double-integer variables

Double integers are similar to ordinary integers, except that they may be up to ten digits long and they require twice the storage space of single integers. They can be mixed in arithmetic expressions with regular integers or reals, but some loss of precision may occur, depending on the range of the result. Double-integer variable names must end with an at sign (@). A double-integer value can range from -2147483648 to 2147483647. Exceeding this range causes the message

```
?OVERFLOW ERROR
```

to be displayed.

Long-integer variables

Long integers are similar to ordinary integers, except that they may be up to 19 digits long. They can be mixed in arithmetic expressions with regular integers or reals, but some loss of precision may occur, depending on the range of the result. Long-integer variable names must end with an ampersand (&). A long-integer value can range from -9223372036854775807 to 9223372036854775807. Exceeding this range causes the message

```
?OVERFLOW ERROR
```

to be displayed.

Long integers are displayed without leading zeros (with a leading minus sign if negative), followed by up to 19 digits without a decimal point.

Many types of financial programming could profit by using long integers and doing all calculations in pennies. The decimal point could be inserted later when reporting results by using the SCALE function, described later in Chapter 3.

Reals

Reals are any positive or negative number within the allowed range. The allowed range varies depending on whether the real type is single precision or double precision. Unlike integers, reals can have a fractional part. The numbers 3, 33, 3.3, -3.3, 3., -3.0, .3, and -.3 are examples of real constants. A numeric constant with a decimal point is always of type real, even if it has only zeros to the right of the decimal point. For example, 3. is a real constant. Constants with more than nine digits and any constant with either a decimal point or an exponent remain as characters and are not tokenized when program lines are entered into a program.

All numbers within a selectable range are printed in conventional or **fixed-point** notation (also called fixed format) by the PRINT statement. For example, 1, +1, -1., 3.14, 999.999, and -0.2 are real numbers expressed in fixed-point notation. The limits of the range are set by the modifiable reserved variable SHOWDIGITS. The default value of SHOWDIGITS is 7.

The SHOWDIGITS variable controls the binary to ASCII conversion of all numbers output by PRINT, including double or long integers. The default setting will format numbers less than 10^7 and greater than or equal to 10^{-7} in fixed format; this is the nominal precision for single real numbers.

When fixed-point notation does not represent the real number accurately, BASIC converts it to scientific, or E (for exponent) notation. The number .0000001 will print in fixed format, but .00000012 will print in scientific notation (as 12E -6), since a significant digit would be dropped if only seven digits were shown.

You can enter from the keyboard any real number within the allowed range in scientific notation. For example, you could type 5.3E12 to represent 5.3 times 10 raised to the twelfth power. When a number is formatted in scientific notation, trailing zeros are not displayed in the mantissa of the number, even if the result is fewer digits than the value of SHOWDIGITS. Here are examples of fixed-point notation versus scientific notation.

Table 2-2

Fixed-point notation	Scientific notation
300	3E+2 = 3*(10^2)
]320	3.2E2 = 3.2*(10^2)
.44	4.4E-1 = 4.4*(10^-1)
-.033	-3.3E-2 = 3.3*(10^-2)
1000000000000	1E+12 = 1*(10^12)

Apple IIGS BASIC considers a single-precision real whose absolute value is less than 1.5E-45 as equal to zero. Double-precision reals with a value smaller than 5.0E-324 are considered equal to zero.

When BASIC displays a real, it displays SHOWDIGITS digits, excluding any exponent. Any significant digits beyond the value of SHOWDIGITS are rounded off. Leading zeros to the left of the decimal point and trailing zeros to the right of the decimal point are not displayed. The decimal point and SHOWDIGITS digits are always displayed, even if the fraction is zero.

Single-precision reals must be within the range of -1.7E38 to 1.7E38. Double precision reals must be within the range of -1.7E308 to 1.7E308. If you enter a number outside those ranges, the message

```
?OVERFLOW ERROR
```

will be displayed.

Strings

A **string** is a sequence of characters enclosed within quotation marks. String variable names must end with a dollar sign ($). Strings may contain from 0 (the null string) to 255 characters. The number of characters in a string is referred to as its **length.** Strings are not fixed in length; they may grow or shrink as necessary.

Strings must be bounded by quotation marks, but they can not contain quotation marks. For example, the statement

```
PRINT "What does "FOO" mean?"
```

prints

```
What does 0 mean?
```

The quotation marks cause BASIC to assume that FOO is a real variable with a value of 0, since it has not been assigned a value. However, strings can contain single quotation marks; the statement

```
PRINT "What does 'FOO' mean?"
```

will be printed as it appears.

If you need to print quotation marks, you can use the CHR$ function with the ASCII code 34 to produce them. For example, this statement will print with quotation marks;

```
PRINT CHR$(34);"One small step for a man...";CHR$(34)
```

The CHR$ function is described in detail in the section "String Functions" later in this chapter.

When a program is run, all string variables initially contain the null string.

Reserved words and variables

Some names cannot be used for variables because they have special meanings for BASIC. For example PRINT, HOME, and OPEN are reserved words in BASIC; they cannot be used as variable names or line labels.

If you attempt to use a reserved word, you will see the

```
?SYNTAX ERROR
```

message

One group of BASIC reserved words are known as **reserved variables**. The reserved variables are

AUXID@ KBD

EOF	JOYY
ERR	PDL9
ERRTOOL	SECONDS@
ERRLIN	PROGNAM$
FRE	DATE$
	TIME$

Your program can refer to the values of these variables, but you cannot assign values to them. For example:

```
PRINT ERRLIN
```

is a legitimate statement, but

```
ERRLIN=50
```

is not.

You can assign values to BASIC's **modifiable reserved variables**

HPOS	OUTREC
VPOS	PREFIX$
INDENT	PROGRAM$
LISTTAB	SHOWDIGITS

Chapter 4, "Controlling Program Execution," explains how to use assignment statements to change the values of these reserved variables. An alphabetical list of all BASIC reserved variables appears in Appendix C, "Reserved Words."

Arrays

An **array** is an ordered collection of single variables, all of the same type. The name of the whole collection, called the **array name,** can be any legal variable name. The last character of the name determines the type of all the variables in the array.

In addition to the normal types, a special type of array, called a **structure,** is supported by IIGS BASIC. The type character that defines structure arrays is the exclamation point (!), as in DRECORD!. The individual variables in a structure are bytes. A structure variable is treated as a short, unsigned integer in a numeric expression, with values from 0 through 255. Structures may only be defined with the DIM statement (as discussed in the next section) and are not allowed as simple variables.

The individual variables (or elements) within an array are numbered, starting with 0. To refer to any element within an array, you specify the name of the array, followed by the number of the element enclosed in parentheses. For example:

```
PRINT AR(3)
```

displays the contents of element number 3 in the array named AR, and

```
PRINT PRICES(147)
```

displays the contents of element number 147 in the array named PRICES. The numbers in parentheses following the array name are called the array's **subscript.** The subscript specifies one unique element within the array.

An array can have three or even more dimensions. The number of **dimensions** is the number of subscripts needed to specify an individual element within it. A one-dimensional array can be viewed as a single list of variables, one after the other in a line. The variable name for such an array needs only one number to specify each variable. For example, a one-dimensional array named Hdoz with six elements includes the elements

```
Hdoz(0), Hdoz(1), Hdoz(2), Hdoz(3), Hdoz(4), Hdoz(5)
```

Note that the largest subscript value is one less than the total number of elements in that dimension of the array.

In a two-dimensional array, two subscripts are needed to specify an individual element. For example, a two-dimensional integer array named D% might include the elements

```
D%(0,0)      D%(1,0),     D%(2,0),
D%(0,1),     D%(1,1),     D%(2,1),
D%(0,2),     D%(1,2),     D%(2,2)
```

D%(2,1) specifies the element in the third column and second row.

Here are some examples of arrays:

Table 2-3

Array	Type	Dimensions	Element referenced
Fudge%(3,2)	Integer	2	4th column, 3rd row
Addrs@(44)	Double integer	1	44th element
Frotz&(77,0)	Long integer	2	78th column, 1st row
STATS(200)	Real	1	200th element
BIG#(300)	Double real	1	44th element
Mumble$(7,3,1)	String	3	8th column, 4th row, 2nd plane
MYRec!(23,10)	Structure	2	24th column, 11th row

The DIM statement

To create an array, you must first tell BASIC the maximum number of elements and dimensions you want the array to accommodate. To do this, you use a DIM (for dimension) statement. For example, the statement

```
DIM Mind%(7,2,3)
```

creates an array named Mind% with three dimensions, with the first subscript ranging from 0 to 7, the second from 0 to 2, and the third from 0 to 3. The character % specifies that this will be an integer array.

Remember that the list of dimensions for all arrays begin with subscript number 0, so the number of elements in each dimension is always one greater than the greatest subscript value. Thus, the number of elements in the array Mind% is equal to (7+1)x(2+1)x(3+1), or 96.

Before you dimension any large arrays, you must expand the data segment so enough memory is available to accommodate the size of the array. The data segment is expanded by using the CLEAR statement with the size option, as described in Chapter 1 and in more detail in Chapter 7.

More than one array can be defined with a single DIM statement by separating the arrays with commas. For example:

```
DIM Light%(78,9), Bulbs$(2,45), Lanterns#(9,2,8), LY(16)
```

creates an integer array named Light%, a string array Bulbs$, a double-precision array Lanterns#, and a real array LY.

Subscripts can range from 0, which is always the first element of each dimension of the array, to a maximum value of 32767. Subscripts may be any integer or real expression; however the resulting value is converted to an integer before the particular element is actually accessed.

If you assign a value to an array element before defining it with a DIM statement, BASIC automatically creates an array having 11 elements per dimension, with subscripts numbered from 0 to 10. For example, when the statement

```
LET TM$(0,0,0) = 29
```

is executed, BASIC defines an array TM$, just as if the statement

```
DIM TM$(10,10,10)
```

had preceded the LET statement.

If the statement

```
PRINT D&(18,LOOP5)
```

is executed before the array D& is defined, a zero is displayed. Unlike other BASICs, IIGS BASIC only automatically defines the dimensions of an array when an assignment occurs, not when a reference occurs. A dummy zero is returned for all array references, regardless of type, if the array has not been dimensioned.

If the value of a subscript refers to either a nonexistent dimension or a nonexistent element (one that is greater than the highest numbered element in a given dimension), the

```
?BAD SUBSCRIPT ERROR
```

message is given. In the example below, both of the statements after the DIM statement will cause this error.

```
)DIM RealArray(1,2))
)PRINT RealArray(1,3)       (DIM statement did not include 3)
)PRINT RealArray(1,1,0)     (DIM did not create 3 dimensions)
```

Statements

Statements can be used in either immediate or deferred execution. **Immediate statements** have no line number or label, and they are executed immediately when entered. **Deferred statements** have line numbers and they may have a label, and are stored in memory (as part of a program) for later execution.

A list of statements may share one line number. In this case, adjacent statements must be separated by a colon (:). The last statement in the list must end with a return. This is called a **statement list**. For example, the following are both legal statement lists:

```
)Xind=4:Yind=5:IF rval>10 THEN rval=xind/yind ELSE rval=xind*yind

)10 FOR index%=1 TO 7:Avalue(index%)=index%:NEXT index%
```

No deferred line, statement, or statement list may exceed 239 characters. After tokenization, a program line, including the line number and 4 overhead bytes, may not exceed 255 characters plus a label of up to 30 characters.

Expressions

There are three kinds of expressions: arithmetic, string, and logical. An expression can be a single constant or variable, or it can be an elaborate mathematical grouping of operators and operands. **Operators** are symbols representing mathematical operations. **Operands** are the variables and constants that operators work on. For example, in the expression 2+3, the operands are 2 and 3 and the operator is the + symbol.

Arithmetic expressions

The operands of arithmetic expressions can be reals, doubles, integers, double integers, or long integers. There are ten arithmetic operators:

Table 2-3

Symbol	Meaning	Example	Value
+	Unary plus	+5	+5
-	Unary minus	-2	-2
^	Exponentiation	2^4	16
*	Multiplication	4*6	24
/	Division	5/2	2.5
DIV	Integer division	7 DIV 5	1
MOD	Modulo	7 MOD 5	2
REMDR	SANE™ remainder	5 REMDR 3	-1
+	Addition	4+7	11
-	Subtraction	9-2	7

Arithmetic operator precedence

In a simple expression like

```
4+8/2
```

you can't tell whether the answer should be 6 or 8, until you know the order (or **precedence**) to carry out the arithmetic operations. Your Apple IIGS computer gives the answer as 8 because it follows these rules of precedence:

1. Any part of the expression enclosed in parentheses will be computed first, according to the following rules. Sets of parentheses grouped one inside another are evaluated from the inside out. First the innermost set is evaluated, then the second innermost, and so forth. For example, the expression 2*(4+3) is equal to 14, while 2*4+3 equals 11.
2. When the unary minus sign is used to indicate a negative number, for example:

   ```
   -3+2
   ```

   your computer will first apply the minus sign to its operand. Thus -3+2 evaluates to -1. It is perfectly legal to use the unary plus sign, but it is always ignored as an operator. For example, the expression

   ```
   +(-4)
   ```

   evaluates to -4, not +4.
3. After applying a unary plus and minus sign, your computer does exponentiation. The expression

   ```
   4+3^2
   ```

is evaluated by squaring 3, and then adding 4. When there are a number of exponentiations, they are done from left to right, so that

```
2^3^2
```

is evaluated by cubing 2, and then squaring the result.

4. After all exponentiations have been calculated, all of the multiplication and division operations are done, from left to right. The multiplication and division operators have equal precedence. For example:

```
24/6/2
```

evaluates to 2.

5. After multiplication and division operations have been calculated, DIV, the integer division operator, evaluates the integer quotient of the division of the first operand by the second. For example:

```
7 DIV 2
```

evaluates to 3.

6. After DIV operations are calculated, the MOD and REMDR operations are done, left to right. MOD evaluates the integer remainder of the division of the first operand by the second. For example:

```
7 MOD 5
```

evaluates to 2. REMDR, the SANE (Standard Apple Numeric Environment) remainder operator, returns a remainder of the smallest possible magnitude. The SANE remainder function differs from the MOD function; its exact definition is included in Appendix K, "SANE Considerations." MOD and REMDR have equal precedence.

7. All additions and subtractions are done last, from left to right. Addition and subtraction have equal precedence.

Logical expressions

Logical expressions, also called **relational** and **Boolean** expressions, are similar to arithmetic expressions, but use additional operators. Where an example of an arithmetic expression might be 2+2, 2=2 is an example of a logical expression. The value of the first expression is 4, while the value of the second expression is true because 2 does equal 2. Likewise, the value of the logical expression 2=3 is false because 2 does not equal 3.

Since BASIC doesn't understand the meaning of truth as such, but only the value of numbers, true and false have been assigned the integer values 1 and 0, respectively. Thus the expression 2=2 returns an integer value of 1 to represent true; 2=3 returns an integer value of 0 to represent false.

Any arithmetic expression with a nonzero value has a truth value of true. Any arithmetic expression with a value equal to 0 has a truth value of false. For example, the logic expression 2+2 has the truth value of true and returns an integer value of 1.

There are eleven logical operators, as described below:

Table 2-4

Symbol	Meaning	Example	Value
=	Equal to	3=3	True
<	Less than	3<1	False
>	Greater than	7>4	True
<= or =<	Less than or equal to	5<=4	False
>= or =>	Greater than or equal to	8>=5	True
<> or ><	Not equal to	4<>4	False
<=>	Ordered (vs unordered)	4<=>NaN	False
AND	Conjunction	5 AND 0	True
OR	Inclusive disjunction	8 OR 3	True
XOR	Exclusive disjunction	8 XOR 3	False
NOT	Negation	NOT 4	False

Logical operator precedence

The precedence of operators in logical expressions is listed below in order of execution from highest to lowest priority. Successive operators of the same priority are executed from left to right.

```
( )
+ - NOT
^
* /
DIV
MOD REMDR
+ -
> < = >= => <= =< <> >< <=>
AND
OR XOR
```

Here are some additional examples of logical expressions, all true:

```
NOT (4=5)
(3-1) OR (NOT-4)
5<>3 AND (6 OR 4)
(2+2)
-2
-(2+2)
NOT 0 AND 6
2*3 + 3/2
9.3
(3.414) * (1.707 + 3.414)
-(2.6)
3 + -(2.5+7)
3<>23444
33 MOD 7
```

Be careful when writing arithmetic expressions. The expression 3<2 is false, and the expression 2<1 is also false, but the expression 3<2<1 is true! This is because BASIC first tests the expression 3<2, and finds it false. False has the integer numeric value 0. It substitutes the value of 0 for 3<2. The expression evaluator then tests the expression 0<1, which is true. The last truth value found is assigned to the expression result, in this case the integer 1.

It is possible to use logical operators in string expressions. For example, "alpha" < "beta" is true.

The ASCII values of the strings to be compared are tested one character at a time, and the first pair of nonidentical characters determines the ranking of the strings. (See Appendix A "ASCII Characters Codes," for a table of these codes and their numeric values.) While simple comparisons of uppercase letters present no problem, the result of comparing mixed-case letters and digits is less straightforward. In every case, the decision will be based on the ASCII code values.

Here are some examples. These string logical expressions are all true:

```
"A"<"B"
"A"<"AA"
"Z">"Antidisestablishmentarianism"
"Anti"<"Antidisestablishmentarianism"
"A">"0"
"a">"A"
"="<">"
```

The Ordered operator

Triple comparisons, such as >=<, <=>, and <>=, are legal constructs in Apple IIGS BASIC. All the combinations are treated as the same operator, the Ordered test, which tests for a relationship peculiar to IEEE (Institute of Electronics and Electrical Engineers) numerics, as implemented by SANE (SANE is further discussed in Appendix K and described in detail in the *Apple Numerics Manual.*)

Two mathematical concepts are supported by SANE to minimize problems created by representation of numbers by a computer. These concepts are internal representations for ± infinity and the concept **Not a Number,** or **NaN.** Most BASICs do not support representations for these concepts in real numbers. Thus, in IIGS BASIC, when you try to divide by zero, infinity is returned as the result.

The concept of NaN is more complex than infinity, but generally NaNs are generated as the result of impossible or meaningless operations. The simplest example of a NaN is the result of trying to take the square root of a negative number. Some other examples are 0*INF, (+INF)-(+INF), 0/0, and X MOD 0; all of which are meaningless operations.

IIGS BASIC generates a result of NaN if it encounters these and other impossible operations during expression evaluation. If you then assign the expression result to a real variable, that variable will not have a numeric value, but a **concept value** instead. The Ordered test checks to see if both its operands are numeric values, that is that neither operand is a concept value. The <=> operator returns true if both operands are numeric representations and false if either operand is a NaN.

A computer cannot always represent the exact mathematical value of a number. An example of this is the value one-third, (1/3). This number is called a repeating decimal, and it is represented in binary as an approximation of the value 1/3 to a finite amount of precision. Thus 1/3 is represented as .333333333333333333000 when it is calculated by BASIC.

Moreover, when the expression 1/3 is assigned to a single-precision real variable, some precision is lost and the value represented becomes closer to .333333343267440795 than to 1/3. As you can see, the first seven digits are correct, but the eighth and later digits are not, and the value is slightly larger than 1/3.

This loss of precision also occurs if the value is stored in a double-precision variable. The expresssion 1/3 becomes closer to the value .333333333333333331483 when assigned (with the LET statement) to a double-precision real variable.

These differences in precision between an expression result (19 to 20 digits of precision) and the precision of variables (7 and 15 digits) can cause problems if you compare an expression to a variable. Thus, the statement

```
1000 IF A#=1/3 THEN …
```

will not be true, even if A# was assigned the value of the expression 1/3.

The loss of precision caused by assignment to a variable must be taken into account when comparing variables with expression results. You can round an expression result to the precision of a variable with the CONV and CONV# functions to eliminate this type of problem in the logical expressions of your IF statements (see the description of IF in Chapter 8).

Functions

Most of the programs that you will write in BASIC will use a relatively small number of tools to solve a large number of different problems.

For example, many scientific and engineering problems require the use of logarithms or trigonometric functions for their solution. You could probably solve these with the use of tables built into your program, or with some equally tedious means, but IIGS BASIC includes a set of tools, called functions, to make these calculations easier.

A **function** takes one or more expressions, called **arguments**, performs some defined operation on them, and **returns** a single value. A function's arguments are arithmetic expressions, except in string functions (described later in this chapter in the section "String Functions") Arguments are enclosed in parentheses following the function name. The returned value is substituted for the function name in the same way that the value of a variable is substituted for the variable name when used by a program.

A function is not a statement itself, but is used as part of a BASIC statement. Functions simply return values; statements tell BASIC what to do with the value returned by the function.

You can either use the functions built into Apple IIGS BASIC, or you can define and use your own functions. The functions built into BASIC perform certain standard operations, such as trigonometric functions, removing fractions from real numbers, finding the absolute value of a number, and so on.

Values returned by functions have types, just as variables and constants have types. All built-in string functions return strings. Most other functions return numbers of type real, but some functions return integer or double-integer results. For example, CONV%, CONV@, and CONV& return regular integers, double integers, and long integers, respectively. IIGS BASIC allows you to freely mix the numeric functions of all types in numeric expressions without generating any errors. However, mixing functions can cause a loss of precision if used incorrectly.

You can assign a real value (returned by a function) to an integer variable, provided it is within the range of -32768 to 32767, since reals are automatically converted to integers by BASIC for this purpose.

For example, the INT function rounds a fractional number to the next lowest whole number (real):

```
)X=-3.3 : Y=7.95
)PRINT INT(X), INT(Y)
-4      7
```

This is equivalent to using

```
)X=-4 : Y=7
```

When a function is included in an expression, BASIC first returns a value for the function, and then evaluates the rest of the expression using the function result. For example, BASIC treats

```
WIDTH=3.3
A=5*INT(WIDTH)+3
```

as

```
A=5*3+3
```

If you want to use or display the value returned by the function, you must include statements in your program to that effect. For example, if you want to display the sine of e, you must first use the SIN function, and then use a PRINT statement to display the returned value:

```
)E=2.718
)PRINT  SIN(E)
.411038
)
```

String Functions

Not all functions operate on numeric arguments; some of them work on strings. A **string** is a sequence of characters and can exist in three forms: as a string constant enclosed in quotation marks, as the content of a string variable (a variable whose name ends with the $ character), or as the value of a string expression.

A string expression's operands are strings and it returns a string value when evaluated. The only operator allowed within a string expression is the concatenation operator, +, which joins strings together. Here are three examples of string expressions:

Table 2-5

Expression	Value
Message$	Whatever has been assigned to MESSAGE$
Message$+"123"	Value of MESSAGE$ with "123" appended to it
"Ap"+"ple"+" IIGS BASIC"	"Apple IIGS BASIC"

An expression having strings as operands but containing any operators other than + is a logical expression, rather than a string expression. For example, the value returned by

```
MSG1$ > MSG2$
```

is not a string, but an integer with 0 for false or 1 for true.

A string containing zero characters is called a **null string** and has a length of 0. A null string is written "". For example the statement

```
)A$=""
```

assigns a null string to A$.

The names of BASIC string functions that return string values end with the $ character.

The LEN function

LEN returns an integer value equal to the length of the string expression, in the range of 0 to 255. For example:

```
)PRINT  LEN("ABCD")
4
)B$="Farm":PRINT  LEN(B$+"House")
9
)
```

If the string expression contains more than 255 characters, the message

```
?STRING  TOO  LONG  ERROR
```

is displayed.

The STR$ function

STR$ evaluates a given arithmetic expression and returns the value as a string. For example:

```
)PRINT  STR$(25/3)
 8.333333
)PRINT  STR$(100000000000)+"More"
 1E+11More
)
```

The CONV$ function

CONV$ evaluates any type of argument and returns a string result. If the expression result is a real type, the string result will be the string that would have been generated by the PRINT statement. If the argument is a string expression, no conversion is performed.

If the expression result is of type integer, CONV$ always returns a fixed-point formatted string of 1 to 19 digits, regardless of the value of SHOWDIGITS. This treatment of integer results is different from the STR$( ) function. Notice the different output from STR$ and CONV$ in this example:

```
)A&=112233445566778899
)PRINT  STR$(A&),CONV$(A&)
 1.122334E+17  112233445566778899
```

The VAL function

VAL evaluates a given string expression and returns the value as a real or an integer number. For example:

```
)PRINT 10*VAL("1.3E4")
130000
)PRINT VAL("13"+"77")
1377
)
```

If any character of the string expression value evaluated is not a legal numeric character (leading spaces are acceptable), the message

```
?TYPE MISMATCH ERROR
```

is displayed.

If the absolute value of the number represented by the value of the string expression is greater than 1.7E308, the message

```
?OVERFLOW ERROR
```

is displayed.

If the string expression value contains more than 255 characters, the message

```
?STRING TOO LONG ERROR
```

is displayed.

The CHR$ function

CHR$ takes an arithmetic expression as an argument and returns a one-character string corresponding to the ASCII value of the evaluated arithmetic expression. For example:

```
)PRINT CHR$(66.8)
C
)R$="68":PRINT CHR$(VAL(R$))
D
)
```

The value of the arithmetic expression is rounded to the nearest whole number if it is a real. It must be in the range 0 to 255, or the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed. (See Appendix A, "ASCII Character Codes.")

The ASC function

ASC returns the ASCII character code corresponding to the first character of the given string expression. If the string expression value is a null string, then the value -1 is returned. For example:

```
)PRINT ASC("BEEP")
66
)d$="BEEP" : PRINT ASC(d$+"S")
66
)
```

The HEX$ function

HEX$ returns as an eight-character string equal to the hexadecimal (base 16), called **hex,** value of the given arithmetic expression. For example:

```
)PRINT HEX$(780)
0000030C
)PRINT HEX$(-1024)
FFFFFC00
)
```

The value of the given arithmetic expression is rounded down to the nearest whole number if necessary. It must be in the decimal range of -2147483648 to +2147483647; otherwise, the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed.

The TEN function

TEN returns the decimal (base 10) equivalent of the last eight or less characters of the given string expression. The value returned will be in the range of -2147483648 to +2147483647, as a double integer. For example:

```
)PRINT TEN("030C")
780
)PRINT TEN("CCCC")
52428
)PRINT TEN("FFFFCCCC")
-13108
)
```

If the first character in an eight-character hex number is a hex digit 8 or greater, then the result of the function will be negative, since the leftmost bit of the number is a 1. The trailing eight or less characters of the value of the given string expression should represent a hex value.

The characters of the string are scanned from last to first and digits and the letters A through F (or a through f) are considered part of the hex string to be converted until a nonhex digit is encountered. The hex string may have zero to eight hex digits. If no hex digits are found in the string, the value zero is returned.

The ERRTXT$ function

ERRTXT$ takes an arithmetic expression, in the range of 1 through 255, and returns a string. The text of the string is copied from the error message tables within IIGS BASIC. The argument is the error number of the error message, as defined in Appendix B, "Error Messages." If the value of the argument is larger than the last defined BASIC error number, the text *PROGRAM* is returned. Upon initial release, IIGS BASIC has defined errors 1 through 88.

You can construct a string exactly like a BASIC error message like this:

```
1000 LET EM$ = "?"+ERRTXT$(67)+"ERROR"
```

The SPACE$ function

SPACE$ returns a string of spaces with the length given by the argument, an arithmetic expression. The value of the expression must be in the range of 0 through 255. If the value is zero, a null string is returned.

The REP$ function

REP$ returns a string composed of the first character of the given string argument, repeated the number of times given by the second arithmetic argument. For example:

```
)PRINT REP$("-",40)
----------------------------------------
)
```

The value of the arithmetic expression must be in the range of 1 through 255, or the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed.

The PFX$ function

PFX$ is a string function that returns a string with the value currently assigned to the ProDOS® 16 prefix given by the numeric argument. The argument must be in the range of 0 through 8. ProDOS 16 supports prefixes 0 through 7, and PFX$ will return the read-only boot-volume name for PFX$(8).

PFX$ will return a null string if the requested prefix is not defined.

The UCASE$ function

UCASE$ is a string function that shifts all the lowercase letters (a through z) in its input string argument to uppercase (A through Z) and returns the string result.

The LEFT$ function

LEFT$ returns a string composed of the leftmost characters of the given string expression. The length of the string returned is defined by an arithmetic expression that immediately follows the string expression in the LEFT$ argument list. For example:

```
)PRINT  LEFT$("Appleskin",5)
Apple
)PRINT  LEFT$("Sparkling",3)
Spa
)
```

If the value of the arithmetic expression exceeds the length of the string expression value, all the characters of the string expression value are returned.

If the string expression value contains more than 255 characters, the message

```
?STRING TOO LONG ERROR
```

is displayed. The value of the arithmetic expression is rounded down to the nearest whole number if necessary. It must be in the range of 1 to 255, or the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed.

The RIGHT$ function

RIGHT$ returns a string composed of the rightmost characters of the given string expression. The length of the string returned is defined by an arithmetic expression that immediately follows the string expression in the RIGHT$ argument list. For example:

```
)PRINT RIGHT$("Appleskin" + "Ware", 8)
skinWare
)B$=RIGHT$("Fruitbat", 3) : PRINT B$
bat
)
```

If the value of the arithmetic expression exceeds the length of the string expression value, all the characters of the string expression value are returned.

If the string expression value contains more than 255 characters, the message

```
?STRING TOO LONG ERROR
```

is displayed. The value of the arithmetic expression must be in the range of 1 through 255, or the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed.

The MID$ function

MID$ returns a substring of a given string expression. You must specify exactly where in the value of the string expression the substring should begin by following the string expression with an arithmetic expression. For example, if you want to retrieve the substring *keeping* from the string *Bookkeeping,* use

```
PRINT MID$("Bookkeeping",5)
```

because the first character in *keeping* is the fifth character in *Bookkeeping*.

You may optionally specify the exact number of characters to be retrieved from the string expression value by including a second arithmetic expression. For example, if you only want the four-character substring *keep* from *Bookkeeping,* use

```
PRINT MID$("Bookkeeping",5,4)
```

because *keep* is four characters in length.

If the value of the first arithmetic expression exceeds the length of the string expression value, then a null string is returned. If the value of the second arithmetic expression specifies a greater number of characters to be retrieved from the string expression value than exist, all of the characters from the position specified by the value of the first arithmetic expression to the end of the value of the string expression are returned. For example:

```
PRINT MID$(A$,255,255)
```

will display one character if the length of A$ is equal to 255; otherwise, a null string is displayed.

If the string expression value contains more than 255 characters, the message

```
?STRING TOO LONG ERROR
```

is displayed. If the value of either arithmetic expression is outside the range of 1 through 255, then the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed.

The program below is an example of the use of MID$ function. Try to figure out what it will do, and then run it:

```
15 A$="ABCD"
25 FOR Loop1 = 1 TO 4
35 FOR Loop2 = 1 TO 4
45 PRINT MID$(A$,Loop1,Loop2)
55 NEXT Loop2
65 NEXT Loop1
```

The INSTR function

INSTR returns the position of the beginning of a specified substring within a given string. For instance, if you want to know where the substring *ai* is in the string *Rain in Spain on the plain,* use

```
)PRINT INSTR("Rain in Spain on the Plain", "ai")
2
)
```

The first occurrence of *ai* is at character position 2. The substring can be the value of any string expression. Note that *ai* occurs at two other places within the string. To find their positions, you must include an optional arithmetic expression after the string expression to begin the string search at a position other than its first character. For example:

```
)PRINT INSTR("Rain in Spain on the plain", "ai", 9)
11
)
```

The arithmetic expression (9 in the example) specifies the character position at which the search should begin. If no arithmetic expression is specified, the search begins with the first character position of the string expression. If the substring is not found within the string expression, the value 0 is returned.

If the arithmetic expression is greater than the length of the string expression or less than 1, then the message

```
?ILLEGAL QUANTITY ERROR
```

is displayed.

The SUB$ function

SUB$ lets you replace any part of a string with a specified substring. The string to be changed can be any string variable, and the substring may be the value of any string expression. You must specify the first character in the string to be changed by following the string with an arithmetic expression. For example:

```
)F$="Hardware" : SUB$(F$,1)="Soft" : PRINT F$
Software
)
```

In this example, the new string *Soft* replaces part of the string *Hardware* contained in the string variable F$. The replacement begins at the first character of F$, and continues until all the characters of the substring *Soft* have been placed in position.

You may optionally include a second arithmetic expression to specify the number of characters in the substring to be used in changing the original string. For example:

```
)F$="Hardware" : B$="Soft" : SUB$(F$,1,2)=B$ : PRINT F$
Sordware
)
```

Here are some additional examples of the use of the SUB$ function:

```
)A$="ABCDEFG":B$=A$:C$=B$:D$=C$:E$=D$:F$=E$
)SUB$(A$,3)="**":PRINT A$
AB**EFG
)SUB$(B$,3,1)="**":PRINT B$
AB*DEFG
)SUB$(C$,3,100)="**":PRINT C$
AB**EFG
)SUB$(D$,3)="**********":PRINT D$
AB**********
)SUB$(E$,3,9)="*********":PRINT E$
AB*********
)SUB$(F$,3,2)="*********":PRINT F$
AB**EFG
```

Numeric functions

Numeric functions may be used in either immediate or deferred execution. The argument to all numeric functions must be an arithmetic expression, except for the SCALB function. Two additional functions, ANU and COMPI, are described in Chapter 8.

All floating-point arithmetic in GS BASIC is done with 64-bits of precision using SANE). This sets limits on the accuracy of the results returned by numeric functions. For most work, the potential rounding off errors generated will not be a problem (or even detectable), but you should be aware that there is a limit. More information can be found in Appendix K, "SANE Considerations," and in the *Apple Numerics Manual.*

The SIN function

SIN returns the sine of an angle given in radians. For example:

```
)PRINT SIN(2.718)
.411038
)
```

The COS function

COS returns the cosine of an angle given in radians. For example:

```
)PRINT COS(1.571)
-2.03673E-04
)
```

The TAN function

TAN returns the tangent of an angle given in radians. For example:

```
)PRINT TAN(3.141)
-5.92653E-04
```

The ATN function

ATN returns the arc tangent, in radians, of the given argument. The value returned represents an angle in the range of -pi/2 to +pi/2 radians. For example:

```
)PRINT ATN(.3456)
.33275
)
```

The INT function

INT returns the largest whole number value less than or equal to the argument value. For example:

```
)PRINT INT(3.3)
3
)X=INT(-3.3) : PRINT X
-4
)
```

Notice that we said *whole number,* not *integer*. This is because the INT function actually returns a real number (note the decimal point and trailing 0 in the examples above).

The RND function

RND returns a random real positive number less than 1. It generates a new random number each time it is used if the argument value is greater than zero.

If the argument value is negative, RND generates the same random number each time it is used with the same argument. If a given negative argument is used to generate a random number, then subsequent random numbers generated with positive arguments will follow the same sequence each time. A different random sequence is initialized by each different negative argument. This is particularly helpful in debugging programs that use RND.

If the argument value is 0, RND returns the most recent previous random number generated (the CLEAR and NEW statements do not affect this). Sometimes this is easier than assigning the last random number to a variable in order to save it. For example:

```
)10 INPUT X : PRIT RND(X);" ";RND(X);" ";RND(X) : GOTO 10
)RUN
?3
.73643  .21479  .53754
?3
.23458  .65986  .54193
?-4
.95754  .95754  .95754
?0
.95754  .95754  .95754
?
?BREAK IN 10
)
```

For example, to get a random whole number, between 50 and 100 inclusive, combine the INT and RND functions in one expression:

```
PRINT  INT(RND(8)*51)+50
```

The SGN function

SGN returns -1 if the argument value is negative, 0 if the value of the argument equals 0, and 1 if the argument value is positive. For example:

```
)PRINT  SGN(-234)
-1
)PRINT  SGN(2496+234)
1
)PRINT  SGN(5E4-5E4)
0
)
```

The ABS function

ABS returns the absolute value of the argument; in other words, the value of the argument if it is positive, 0 if the value is zero, and the negative of the argument value if it is negative. For example:

```
)PRINT  ABS(345)
345
)PRINT  ABS(24-363)
339
```

The SQR function

SQR returns the positive square root of the argument value. For example:

```
)PRINT  SQR(3^2+4^2)
5
)
```

The EXP, EXP1, and EXP2 functions

EXP raises e (to 6 places, e=2.718282) to the power indicated by the argument value. For example:

```
)PRINT  EXP(3)
20.0855
)
```

EXP1 raises e to the power indicated by the argument value minus one. EXP1(x) accurately computes e^x-1. If the input argument x is small, then the computation of EXP1(x) is more accurate than the straightforward computation of e^x-1 by exponentiation and subtraction.

EXP2 raises 2 to the power indicated by the argument value.

The FIX function

FIX returns the integral portion of the value of the argument, truncating the fraction of the absolute value of the argument. FIX differs from INT in that FIX does not return the next lower number for a negative argument. FIX is equivalent to the statement SGN(x)*INT(ABS(x)). For example:

```
)PRINT  FIX(3.333),FIX(-3.333)
 3-3
)
```

The LOG, LOG2, and LOGB% functions

LOG returns the natural (base e) logarithm of the argument value. For example:

```
)PRINT  LOG(20.08553)
3
)
```

LOG1 returns the natural logarithm of one plus argument value. LOG1(x) accurately computes LOG1(1+x). If the input argument x is small, then the computation of LOG1(1+x) is more accurate than the straightforward computation of LOG1(1 + x) by adding x to 1 and taking the natural logarithm of the result. For example:

```
)PRINT  LOG1(20.08553)
3.048587
)
```

LOG2 returns the base 2 logarithm of the argument value.

LOGB% returns the binary exponent of the argument value as a signed integer.

The NEGATE function

Negate returns the value of the negative arthmetic expression. This seemingly simple function is included due to the presence of the specialized SANE data type representations for infinity and NaN results, which are returned as the result of a meaningless calculation, such as -0/+0. You should use NEGATE rather than -1* the arithmetic expression to properly negate floating-point values.

The ROUND function

ROUND returns the integral value nearest the value of the arithmetic expression, according to the rounding direction of SANE settings. ROUND should be used in place of the common INT(arithmetic expression+.5), because it will return a result consistent with the other capabilities of SANE.

The SCALB function

SCALB is a two-argument function whose first argument, a single-integer expression is used as a base 2 scale factor for the value of the second argument. SCALB scales the arithmetic expression by 2 to the power of the single-integer expression, effectively returning the operand shifted left or right in the binary places specified by the single-integer expression. For example:

```
)PRINT SCALB(4,10): REM 10 * 2^4 = 160
160
)
```

LOGB% is related to SCALB, returning the single-integer expression for a given arithmetic expression.

The CONV& function

CONV& evaluates the given argument and returns a long integer value. For example:

```
)PRINT  CONV&(2178-7954)
-5776
)PRINT  CONV&("4.214")
4
)
```

If the argument is a string, then the effect is the same as using VAL followed, by CONV& (the VAL function is described ealier in this chapter, under the heading "String Functions"). The value returned must be within the range of 922337203685775807 to -9223372036854775808 (-9223372036854775807 from the keyboard), or the message

```
?OVERFLOW  ERROR
```

is displayed.

The CONV function

CONV evaluates the argument and returns a real value. The value may be assigned to a regular integer. The conversion from real to integer is automatic in the latter case. If CONV is used with a string expression, the effect is the same as with the VAL function. For example:

```
)G&=234234  :  H&=523523  :  PRINT  CONV(H&-G&)
289289
)
```

The CONV@ function

CONV@ evaluates the given expression and returns a double-integer value, rounding off to the nearest whole number. The value must be within the range of -2147483648 to +2147483647. For example:

```
)PRINT  CONV@(123456789.12)
123456789
)
```

The CONV% function

CONV% evaluates the argument and returns an integer value, rounding off to the nearest whole number. The value returned must be within the range of -32768 to 32767, or the message

```
?OVERFLOW  ERROR
```

is displayed. For example:

```
)PRINT  CONV%(423.94)
424
)A&=7656  :  B&=364  :  PRINT  CONV%(A&/B&)
21
)
```

Miscellaneous functions

This section describes some Apple IIGS BASIC functions that cannot be classified as string or numeric, These miscellaneous functions have a wide range of applications. Some use arguments not allowed in most functions, and some allow optional arguments.

Other BASIC functions are described in Chapter 7, "Advanced Topics," or in Chapter 8, "BASIC Reference."

The BTN function

BTN returns the state of the three sense inputs ($E0C061, 62, and 63) as 0 or 1. Various devices, such as the buttons on paddles or joysticks and the Apple and Option keys, can control the state of these inputs.

The FILE function

FILE returns the value 1 if the file given by the first argument, a pathname string expression, exists or the value 0 if the file does not exist. If any error other than FILE NOT FOUND is encountered that error will be displayed. If the optional second argument, FILTYP= filetype, is not specified, a result of true will be returned for any file type. The syntax for the FILE function is:

```
FILE(sexpr[,FILTYP= TXT'SRC'BDF'ubexpr)
```

Where **sexpr** is a string expression and **ubexpr** is an **unsigned byte** expression.

If the file type that you specify is not the same as the file's actual type, a

```
?FILE TYPE ERROR
```

message will be displayed.

The reserved variable AUXID@ will contain the subtype from the directory entry of the file. The function FILTYP(0) will return the file type of the file.

The JOYX and JOYY variables

JOYX is a function that reads two of the four game paddle inputs (if they are plugged in) specified by the integer argument. The integer expression must result in a number from 0 to 2, otherwise an

```
?ILLEGAL QUANTITY ERROR
```

message is displayed. JOYX returns the value for the paddle given by the expression.

In addition, the reserved variable JOYY is set with the value of the paddle **unsigned byte expression**+1 when the JOYX function is called. This function eliminates the interaction between paddles due to the coupling of the hardware one-shot timers by timing both paddles in parallel. Both JOYX and JOYY return a result with 8 significant bits.

The PDL function

PDL reads the position of the game control paddle (if it is plugged in) given by the argument, a number from 0 to 3. PDL returns a value in the range of 0 through 255. The reserved variable PDL9 will return the 9-bit result calculated by the prior execution of the PDL function.

The PEEK function

PEEK reads a byte from memory at the address given by the integer expression and returns an unsigned integer in the range of 0 through 255. The integer expression must be a positive integer less than 2^{24}. Care should be exercised in using PEEK, since improperly reading many I/O devices and control registers can crash the system.

Programmers concerned about writing programs that will run on newer versions of the Apple II product family should avoid the use of the PEEK function because it contains a hard-coded address that may not be supported in the future. PEEK is provided for those who want to build nontransportable, locked-in programs.

Defining your own functions

You can define your own functions to perform operations that are not provided by any of BASIC's built-in functions. Once a function is defined, it is also available in immediate execution, as long as the defining program is still in memory and the program containing the DEF FN statement has been executed but has not been edited or expand with new lines. A user-defined function may not be defined in immediate mode.

If your program defines a user function and a CHAIN statement calls a new program, the old functions are discarded and the new program may not use them. If you try to do this an

```
?UNDEF'D PROC/FUNCTION ERROR
```

message will be displayed.

The DEF FN statement

The DEF FN statement is used to let the user define simple functions. The name of the function and its parameters must follow the reserved word FN. The parameter list consists of one or more simple variables enclosed in parentheses and separated by commas. Parameters named in a DEF FN statement are isolated from variables with the same names used elsewhere in a program. The statement must conclude with a **defining expression.**

Simple functions can be of any type, including string functions and a function can have one or more parameters, including string parameters. The parameter variables are allocated in a separate, temporary local symbol table when a function is used and discarded after the expression is evaluated. The syntax for a simple function definition is

```
      DEF FN name (paramlist) = arithmetic expression
or    DEF FN name% (paramlist) = arithmetic expression
or    DEF FN name@ (paramlist) = arithmetic expression
or    DEF FN name& (paramlist) = arithmetic expression
or    DEF FN name# (paramlist) = arithmetic expression
or    DEF FN name$ (paramlist) = string expression
```

The defining expression may be any legal arithmetic or string expression, as shown above. The defining expression may contain the parameters as well as any other integer or real variables. The type of the expression must match the type of the function.

For example, this statement defines a function named RECIP, with X as the argument and 1/X as the defining arithmetic expression:

```
)55 DEF FN RECIP(X) = 1/X
```

Here are some more examples:

```
)10 DEF FN MINUS(X) = -ABS(X)
)20 DEF FN SWORDED.4(C) = INT(RND(3)*100)
)30 DEF FN M5BY7.MAT#(DED#,F%) = DED#*LOG(33)-ABS(F%)
)40 DEF FN ENDS$(A$,B$,S%)=LEFT$(A$,S%)+RIGHT$(B$,S%)
```

In addition to the single-expression functions described here, GS BASIC also supports multiline numeric (but not string) functions and procedures. These advanced programming tools are described in Chapter 7, "Advanced Topics."

Using a defined function

Once a simple function has been defined, you can use it anywhere that an arithmetic expression can be used. After you enter function definitions in a program, the functions are still not available to be used until the program is run.

When a RUN or CHAIN statement is executed, the entire program is scanned for DEF statements (referred to as a **DEF scan**) and a special entry is made in the variable table for each DEF function in the program. This entry contains a pointer to the parameter list in the DEF statement; the pointer is used when the function is referenced.

Using (or referencing) a function requires that its name be preceded with the reserved word FN. Here is an example of how you might use the Minus function:

```
)20 A = FN Minus(2)
```

In this example, the real variable A is assigned the value -2.

Here is another example:

```
)30 PRINT 4*FN Minus(A)*3
```

If the value of A was -2, then -24 would be displayed on the screen.

Remarks about defining functions

The defining expression used to define a function can refer to any real or integer variable and any or all of the parameters. For example, consider the following sequence:

```
)5 B=3
)10 DEF FN Foo(A)=A*C
)15 PRINT FN Foo(B) : REM displays 0 because C is undefined
)20 C=5 : REM C now exists and equals 5.0
)25 PRINT FN Foo(B) : REM displays 15
```

The parameters to the function need not appear in the defining expression. In this case, the function's argument, B, is ignored in evaluating the expression.

When a function is referenced, the value of an argument is assigned to the parameter in the parallel position. Even if a parameter is not used in the defining expression, the argument in a reference will be evaluated and assigned to the parameter, so it must be something legal.

For example the formal argument Zilch in the next example is real, even though it is not used in the expression:

```
)10 DEF FN BAZ(Zilch)=2+2
```

Functions may not be defined more than once in a program.

If a function is used in an immediate statement before the program containing the DEF FN statement is run, the

```
?UNDEF'D PROC/FUNCTION ERROR
```

message is displayed.

The entry made by the DEF scan (of a RUN or CHAIN statement) for each DEF FN statement uses 1 byte per character in the function name, 2 overhead bytes, and 6 bytes for information about the function as follows: a byte containing the parameter count, 3 bytes for program relative offset, and 2 bytes for the DEF statement line number.

Chapter 3

Entering, Displaying, and Formatting Data

Displaying text on the screen xx

The PRINT statement xx
TAB and SPC specifications xx

Entering data xx

The INPUT statement xx
- Entering numbers xx
- Quotation marks and commas in string input xx
- Null strings xx

The GET$ Statement xx

Storing data within your program xx

The DATA and READ statement xx
The RESTORE statement xx

Formatting information xx

The PRINT USING and PRINT# USING statements xx
The IMAGE statement xx
Output format specifications xx
- String specs xx
- Literal specs xx
- Numeric specs xx
  - The Fixspec xx
  - The Scispec xx
  - The Engrspec xx
- The SCALE function xx

This chapter describes the tools that are provided by Apple IIGS BASIC for entering, displaying, and formatting data.

Your computer's keyboard and display screen are collectively termed the **console.** Console I/O statements allow you to control when, where, and how specific characters are displayed on the screen or read from the keyboard.

The screen is divided by five 16-character-wide tab fields. If you have typed or printed a character at the last character position of tab fields one through four, tabbing will move the cursor to the beginning of the second field following that character.

If the text window is less than 80 characters wide, tabbing may send the cursor down one or more lines.

Displaying text on the screen

The PRINT statement is used to display text on the screen. TAB and SPC specifications provide further control over the appearance of the display.

The PRINT statement

To display text on the screen, follow the reserved word PRINT with a list of the items that you want to appear. The item list can include any expression, comma, semicolon, or TAB or SPC specification. (TAB and SPC are discussed in the next section.)

BASIC evaluates the expressions in PRINT statements and displays their values in sequence. A PRINT statement without an item list causes the cursor to move to the beginning of the next screen line.

If a comma separates two expressions, a tab space separates their values on the screen; if a semicolon separates them, the second value is displayed after the first, with no intervening spaces. For example:

```
)PRINT "In";"divisible"; 543, 598/754.42
Indivisible543  .792662
)
```

You can follow the last expression in a PRINT statement with a semicolon, comma, or nothing. If there is nothing after the last expression, the cursor moves to the beginning of the next screen line. A comma causes the cursor to move to the next tab field, and a semicolon leaves the cursor in the position immediately following the last character displayed (suppressing carriage returns).

Here are a few examples of PRINT statements:

```
)PRINT "X"
X
)X&=3 : PRINT X&
3
)PRINT 10, 20
1020
)PRINT 10;" "; 20
1020
)A$="Apple" : PRINT A$
Apple
)A$="Johnny" : B$="Apple" : C$="seed" : PRINT A$;" ";B$;C$
Johnny Appleseed
```

Warning:

If you don't use either commas or semicolons to separate expressions in a PRINT statement, BASIC will attempt to figure out where one expression ends and the next one begins. If it succeeds, the effect is the same as using a semicolon. If it doesn't succeed, either a

```
?SYNTAX ERROR
```

message appears, or the wrong values are displayed. You should use the semicolon to avoid confusion.

❖ *Note:* The statement

```
)PRINT A$+B$
```

will give

```
?STRING TOO LONG ERROR
```

message if the combined length of the two strings is greater than 255 characters. However, you can display the apparent combined string using

```
)PRINT A$;B$
```

without worrying about its length.

When you are entering PRINT statements into a program, you can use a question mark instead of the reserved word PRINT (it lists as PRINT). You don't save any memory by using abbreviations, only typing time.

TAB and SPC specifications

You can insert spaces into text displayed by a PRINT statement by putting a TAB or SPC specification into the PRINT statement.

A TAB or SPC can be inserted immediately before or after any expression, comma, or semicolon in a PRINT statement's expression list.

TAB and SPC are followed by an arithmetic expression enclosed in parentheses. The integer value of the expression following the word TAB defines the number of spaces from the left margin of the text window to begin printing text. If you specify an expression that is less than the number of the current print position, no spaces will be inserted before the next character to be printed.

The integer value of the expression following the word SPC defines the number of spaces to be inserted after the character last printed. Examples of the use of TAB and SPC are given below.

```
)PRINT "Great"; TAB(8); 347
Great 347
)PRINT "Underhanded"; TAB(8); 553
Underhanded553
)PRINT "A"; SPC(1); "B"; SPC(2); "C"
A B C
)PRINT "D"; TAB(5); "E"; SPC(5); "F"
D   E   F
```

Each SPC statement is limited to a maximum value of 255, but you can insert as many spaces as you need by stringing together a series of SPC statements like this:

```
)SPC(250)SPC(139)SPC(255)
```

❖ *Note:* The PRINT ... USING statement gives you greater control over the display of text on the screen. See the section titled, "Formatting Information" later in this chapter.

Entering data

You use the INPUT statement to enter numbers or text and the GET$ statement to assign a keyboard character to a string variable in a program.

The INPUT statement

INPUT accepts numbers or text typed at the keyboard and assigns them to variables specified in the INPUT statement. For example, if you wanted users of your program to input their age in years, you could use

```
20 PRINT "Enter your age in years"
25 INPUT AGE
```

When INPUT is executed in this form, BASIC automatically displays a question mark on the screen and waits until the user types something. For example:

```
)RUN
Enter your age in years
?38)
```

You may optionally include a string in an INPUT statement, like this:

```
20 INPUT "Enter your age in years"; AGE
```

The optional string must be a sequence of characters in quotation marks, followed by a comma or semicolon; it cannot be a string variable or expression. When you use an optional string, it is displayed exactly as you specified; BASIC does not add a question mark, spaces, or other punctuation after the string. You can include only one optional string.

After the optional string, you must include one or more variable names, separated by commas, like this:

```
1000 INPUT "Please type three words"; A$, B$, C$
```

INPUT expects you to type a number for each numeric variable and a string for each string variable in the INPUT statement. The numbers and strings are expected in the same order in which the variables occur in the statement.

The numbers and strings that you type in response to an INPUT statement must be separated from each other by commas or by pressing Return. If you use Return and another number or string is still expected, BASIC will display not one, but two question marks on the next screen line. The input numbers and strings are assigned to the variables in sequence. For example:

```
1000 INPUT "Please type four numbers: ";A,B,C,D
```

This statement expects you to either enter four numbers separated by commas like this:

```
100, 200, 300, 400 (followed by Return)
```

or by returns, like this:

```
Please type four numbers: 100
??200
??300
??400
```

You can halt execution during an INPUT statement by pressing Control-C any time before you press Return. BASIC will recognize the Control-C immediately and discard any input entered before Control-C was pressed.

Entering numbers

For arithmetic variables, INPUT will accept only numbers. Remember that the plus sign, space, hyphen, period, and E characters are all legitimate parts of numbers. Any leading or trailing spaces in numbers are ignored, but embedded spaces are not allowed.

Input that is not a legitimate number (such as a string or a return) will cause the message

```
?REENTER
```

to be displayed, and BASIC will reexecute the INPUT statement from the beginning.

If an input number is not the same type as the corresponding variable, it will be automatically converted to the same type; if necessary, the number will be rounded. For example:

```
)5 INPUT G%
)10 PRINT G%
)RUN
?6.87
7
)
```

Quotation marks and commas in string input

Quotation marks and commas in string input are handled differently, depending on the position of the string variable in the INPUT statement to which the string will be assigned.

If the string variable is the last (or only) variable in the INPUT statement, any quotation marks or commas are treated as ordinary characters in the string.

If the string variable is not the last variable in the INPUT statement, the comma and quotation mark characters are treated specially. The comma separates strings in the same manner as it separates numbers. By enclosing a string in quotation marks, you can include commas in it without affecting the spacing. The quotation marks are not considered part of the string.

The closing quotation mark of a string can be omitted if the Return key is used to end the string. If a quotation mark is not the first character typed, then a quotation mark can be included anywhere else in the string without being treated specially. Here are some examples of the effects of quotation marks: (remember that a program can be halted by typing Control-C as response to an INPUT statement):

```
)10 INPUT X$, Y$ : PRINT X$, Y$ : GOTO 10
)RUN
?is, is
is is
?"is, "is is, "is
?REENTER
?"is", 'is'
is 'is'
?is", is"
is"  is"
?""is, ""is
?REENTER
?is"", is""
is"" is""
?"is"", "is""
?REENTER
?"is,isn't", "is,isn't"
is,isn't     "is, isn't"
?
?PROGRAM INTERRUPTED IN 10
)
```

Null strings

If the user simply presses Return key, without typing any characters, when a string response is expected, BASIC interprets the response as a null string.

❖ *Note:* Pressing the Control and Option keys in combination with a letter key will pass the control code character to BASIC as an input character.

The GET$ statement

GET$ is used to assign a single alphanumeric character from the keyboard to a specified string variable in your program, without displaying it on the screen and without requiring that the Return key be pressed. The specified variable must follow the reserved word GET$. For example:

```
)100 PRINT "Press any key";
)110 GET$ PRESS$
)120 PRINT ". You pressed the ";PRESS$;" key!"
)130 GOTO 100
```

Note that GET$, unlike the GET statement in Applesoft BASIC, does not allow you to use a numeric variable; you can only use a string variable with GET$.

If you want to use GET$ to input numbers, you must get a string variable, and then convert the resulting string to a number with the VAL function (as described in the "String Functions" section of Chapter 2). In most cases, it is more convenient to use the INPUT statement for number.

You can only use the GET$ statement with deferred execution.

Warning:

If you display the value of a string variable that contains a control character, that control character can affect operation of the .CONSOLE device driver. For example, if you pressed Control-L as a response in the program example above, the text screen would be cleared when line 120 displayed PRESS$.

❖ *Note:* If a program that uses GET$ is called by an EXEC file, the input for the GET$ statement will be taken from the EXEC file instead of from the keyboard.

Storing data within your program

The DATA and READ statements allow you to store data within a program itself. The RESTORE statement can be used in conjunction with the DATA statement. Although these are not actually console I/O statements, they are included here because their effect is similar to that of INPUT and GET statements.

The DATA and READ statements

The DATA statement is used to provide a list of **data elements** to be read by a READ statement. DATA statements do not have to precede READ statements in a program. Data elements can be strings, reals, integers, double integers, and long integers. String data elements do not have to be bounded by quotation marks. For example:

```
)1220 DATA WHEN, 5, -4, "EQUALS", 1.000
```

The DATA statement can only be used in deferred execution, or an

```
?ILLEGAL DIRECT ERROR
```

message appears.

A **data element list** consists of all of the elements in all of the DATA statements in a program. READ statements are used to read from a data element list and to assign the values to variables. When BASIC executes the first READ statement in a program, it assigns the value of the first data element in the list to the first variable in the READ statement. The second variable in the READ statement (if there is one) is assigned to the second element in the data list, and so on. For example:

```
)30 READ A$, B%, C&, D$, E
)40 PRINT A$;" ";B%; C&;" ";D$;" ";E
)500 DATA When, 5, -4, "Equals", 1.000
)RUN
When 5-4 Equals 1.000
)
```

Data elements assigned to arithmetic variables generally follow the same rules that numbers assigned to arithmetic variables by INPUT statements follow.

If you use Control-C as a data element, it does not halt execution of the program, even when it is the first character of an element. With this exception, data elements read into string variables follow the rules for INPUT responses assigned to string variables:

- Either literal or quotation-mark-enclosed (quoted) strings may be used.
- Quotation marks appearing within a quoted string cause the

  ```
  ?SYNTAX ERROR
  ```

  message, but all other characters, including commas, are accepted as characters in that string.
- The colon and comma are accepted only in quoted strings.
- Control-M (the Return character) is never accepted as a data element.

If a READ statement attempts to assign a string data element value to an arithmetic variable, a

```
?SYNTAX ERROR IN 99999
```

message appears, where 99999 represents some valid line number.

In reading a data element, BASIC assigns a value of zero or null string (depending on the variable's type) under any of the following conditions:

- A comma is the first character (excluding spaces) following the reserved word DATA.
- There is no data element between two commas.
- The last character in a DATA statement is a comma (when the comma is being read as a data element).

For example when this statement is read:

```
100 DATA ,
```

it can result in up to two element assignments consisting of zeros or null strings.

When variables in a READ statement have been assigned values from the data element list, BASIC leaves a **data list pointer** immediately following the last element read. The next READ statement executed (if any) begins using the data list from the pointer position. A RUN, CLEAR, or RESTORE statement moves pointer back to the first element in the data list.

When all of the elements in a DATA statement have been read, the pointer moves on to the next DATA statement with a higher line number, and reading continues with the first element listed. An attempt to read past the end of the data list produces an error message. For example, you might see

```
?OUT OF DATA ERROR IN 3400
```

When 3400 is the line number of the READ statement that asked for additional data.

In immediate execution, you can only read elements from DATA statements in a program that is currently in memory that is, one that has been typed in, loaded, or run. If no DATA statement is in memory, the message

```
?OUT OF DATA ERROR
```

is displayed. Execution of a READ statement does not reset the data list pointer back to the first element in the data list after BASIC reads the last element in the list.

❖ *Note:* You cannot follow DATA with any executable statements on the same program line. Anything following a DATA statement until the next line number is considered to be part of a data list.

The RESTORE statement

RESTORE moves the data list pointer back to the beginning of the data list. This allows you to read the same data more than once. You can use the optional parameter, a line number or label, to set the data list pointer to the beginning of a DATA statement anywhere in a program. The line number or label you use must be the line number of an existing line; otherwise, the message

```
?UNDEF'D STATEMENT ERROR
```

will be displayed. After a RESTORE line number is executed, the data element list is read forward through all the DATA statements with the given and higher line numbers.

Formatting information

Apple IIGS BASIC provides several tools for formatting its output. You can use the PRINT USING and PRINT# USING statements with a variety of specifications to produce different formats.

The PRINT USING and PRINT# USING statements

PRINT USING allows you to control the format of information displayed on the screen, and PRINT# USING controls the format of data written to files. We will refer to these statements collectively as PRINT(#) USING. Both statements work with either numeric or string data.

Printing fields, defined by format specifications (called **specs**), outline the way that information is formatted.

Specs can be included with the PRINT(#) USING statement in the form of a string expression, or they may be part of an IMAGE statement elsewhere in the program that the PRINT(#) USING statement references. They appear as codes of letters, numbers, and/or symbols. When the specs are used in an IMAGE statement, the PRINT(#) USING statement must include its line number. Here are some examples of PRINT USING statements.

```
)10 PRINT USING 100; A$, B%, C
)100 IMAGE 6A, 5#, #.6Z4E

)1 B$="6A, 5#, #.6Z4E"
)10 PRINT USING B$; A$, B%, C

)10 PRINT USING "6A, 5#, #.6Z4E"; A$, B%, C
```

All three of these examples do the same thing—they display the values of the variables A$, B%, and C, formatted according to the specs supplied by the user. The types of BASIC format specs and how to use them are described later in this chapter.

Note that the commas in a PRINT (#) USING statement expression list simply serve to separate the expressions they do not cause the cursor to move to the next tab field, as in a PRINT statement item list. However, a semicolon at the end of the expression list suppresses a carriage return, just as it does with PRINT.

If the number of expressions in the expression list exceeds the number of specs as in `)10 PRINT USING`, BASIC will use the spec list again from the beginning until all the expressions have been evaluated and used. A single spec will be used by all expressions in the list. This means that if you have a number of values to be displayed using the same spec, you only have to write the spec once.

The following errors can occur when a PRINT(#) USING statement is executed:

If the USING clause references an IMAGE statement, and the IMAGE statement does not exist, the

```
?UNDEF'D STATEMENT ERROR
```

message is displayed.

- If the USING clause references a string variable or contains a string, and the string value is null, a

```
?SYNTAX ERROR
```

message appears. An IMAGE statement that does not contain specs will result in the same error message. When a syntax error occurs in an IMAGE statement, the message gives the line number of the PRINT(#) USING statement, not that of the IMAGE statement.

- If the type of an expression does not match the type of its spec (for example a string expression with a numeric spec) a

```
?TYPE MISMATCH ERROR
```

message is displayed.

The IMAGE statement

An IMAGE statement contains a sequence of specs separated by commas. Each spec corresponds to an expression in a PRINT(#) USING statement, and controls the printed or displayed format of the expression value. (An exception is the literal spec, which does not correspond to an expression.)

You can use the IMAGE statement only with deferred execution, and it must be the only statement on the line. Here is an example of an IMAGE statement with three specs:

```
10 IMAGE +5#.3#,10X,-#.5#4E
```

The following example show the output generated by the IMAGE statement in line 10 with two specs:

```
10 IMAGE +###.###, 3"."
100 PRINT USING 10; 1.5, 3.14159, 172.9, 5
)RUN
+ 1.500...+ 3.142...+172.900...+ 5.000...
)
```

Output format specifications

There are three kinds of format specs:

- **String specs** control the format of string values in a PRINT(#) USING statement.
- **Literal specs** insert one or more spaces, line returns, or copies of a specified string into the text displayed by the PRINT(#) USING statement.
- **Numeric specs** control the format of a numeric value displayed by a PRINT(#) USING statement. Numeric specs can be fixed-point, scientific notation, or engineering notation.

String specs

A string spec defines the field format and width for a string value, and specifies whether the string value is to be left-aligned, right-aligned, or centered within the field.

The codes are as follows:

A Left-aligned
R Right-aligned
C Centered

If the string has fewer characters than the field allows, the empty positions are filled with spaces.

You can set the width of the field either by specifying the number of characters to be used in the field, or by preceding the spec with a positive integer equal to the length in characters of the field. For example, a six-character, right-aligned field could be defined either by RRRRRR or 6R. The specs 9C and CCCCCCCCC produce the same result: a nine-character field with its text centered in the field. These numbers are called **repeat factors.** a repeat factor can be any positive integer from 1 through 255. It affects only the single character immediately following it. (Thus, 5AA or A5A means the same thing as 6A. A repeat factor greater than 255 causes an

```
?ILLEGAL QUANTITY ERROR
```

message.

Here's an example of using string specs to format string output into three columns:

```
10 IMAGE 15A, 15C, 10R
15 PRINT
100 PRINT USING 10;"COMPOSER", "TITLE", "KEY"
200 PRINT USING 10; "GRINSZ", "SONATA FOR HARP","F SHARP"
300 PRINT USING 10; "RIBBITT", "WATER SONG", "E FLAT"
400 PRINT
```

The IMAGE statement in line 10 defines three fields: a 15-character field with a left-aligned string in it, a 15-character field with a string centered in it, and a 10-character field with a right-aligned string in it. Note that these add up to 40 characters, so they will fill one half of a line on the screen. This program runs as follows:

```
)RUN

COMPOSER        TITLE                KEY
GRINSZ          SONATA FOR HARP      F SHARP
RIBBITT         WATER SONG           E FLAT
```

If a string value exceeds the length of its field specification, BASIC truncates it. For example,

changing the string spec in the program fragment above to

```
)10 IMAGE 5A, 5C, 5R
```

has the following effect:

```
)RUN

COMPOTITLE KEY
GRINSSONATF SHA
RIBBIWATERE FLA

)
```

Literal specs

A literal spec does not format the value of an expression; instead, it inserts spaces, line returns, or a fixed string into the output. The codes are as follows:

X	Prints a space
/	Prints a line return
"	Encloses a literal string to be printed

For example, the spec

```
4X
```

inserts four spaces into the output, and the spec

```
2/
```

inserts two line returns.

When you place a repeat factor in front of a literal spec string, it affects the entire string. For example, the spec

```
3"AB"
```

inserts

```
ABABAB
```

Separate spec values are necessary for each type of insertion. For example, two spec values are needed to insert three spaces followed by five asterisks:

```
3X,5"*"
```

Numeric specs

You can format a numeric value, regardless of its type, in fixed-point, scientific, or engineering style. There is a separate kind of numeric spec for each of these formats.

All three numeric formats use the following **digit specs:**

#	Reserves one numeric digit position and suppresses leading zeros.
&	Reserves a position for a digit or comma (at least five digit positions must be reserved to the left of the decimal point).
Z	Reserves one numeric digit position and prints leading zeros.

The fixspec: The fixed-point specification, called **fixspec,** controls the format of fixed-point numbers. Fixed-point numbers are any numbers displayed without exponents, including integers, long integers, and real numbers.

In addition to the digit spec characters, the fixspec uses the following characters. (Note that if Z is used, $$, ++, and -- may not be used.)

+	Reserves a character position for number sign
-	Reserves a position for a minus sign if the number is negative
$	Reserves a position for a dollar sign ($)
* *	Prints asterisks instead of leading spaces
++	Reserves rightmost positions for a number sign and dollar sign (if any)
--	Same as ++, except the sign is printed only if the number is negative
$ $	Reserves leftmost unused position for a dollar sign and number sign (if any)

Here is an example of a simple fixspec appearing in a PRINT USING statement:

```
)PRINT USING "+###.###"; 3.14159
+ 3.142
```

You can use repeat factors with all the digit-spec characters. The # character reserves one numeric digit position. Leading zeros (if present) are replaced with spaces. For example:

```
)PRINT USING "+6#.3#"; 09999
+ 9999.000
```

A Z reserves one numeric digit position, just like a #, but prints leading zeros. For example:

```
)PRINT USING "+6Z.3Z"; 09999
+009999.000
```

An & character reserves one position for a numeric digit or comma. Commas are inserted after every third digit left of the decimal point, and they are included in the character count: leading zeros are replaced with spaces. At least five digit positions must be reserved to the left of the decimal point when using &. For example:

```
)PRINT USING "+6&.3&"; 09999
+ 9,999.000
```

The examples above all show a decimal point with digits to the left and to the right. However, you can also specify no decimal point, a decimal point with nothing to the left, or a decimal point with nothing to the right. Remember that integer expressions can have no fractional part. If you specify a fixspec with a fractional part and apply it to an integer expression, only zeros will appear to the right of the decimal point, unless you use the SCALE function (described later in this chapter).

BASIC will round off the value to be displayed, if necessary, to fit the number of digits specified to the right of the decimal point. However, if the number exceeds the number of digits specified to the left of the decimal point, the entire field is filled with exclamation points.

❖ *Note:* You can mix the digit spec characters &, #, and Z in a spec list, but those appearing to the left of the decimal point yield to the character with the highest precedence. Their order of precedence is &, #, Z. If an & appears, the formatted output will have commas inserted and leading zeros suppressed.

In all the examples of fixspecs so far, we have shown a +, which reserves a position for the sign of the number. The + causes the sign to be printed in all cases. A - causes the sign to be printed only if the number is negative; a space is printed with positive numbers. The sign of the number can also follow the last digit.

For financial output, use a $ to reserve a character position for a dollar sign. A pair of asterisks (**) causes asterisks to be printed instead of leading spaces when there are unused digit positions in the output field. For example:

```
)PRINT USING "**+6#.3#"; 09999
+**9999.000
)PRINT USING "**$6#.3#-"; 09999
***$9999,000
)PRINT USING "+$6#.2#"; 09999
+ $9999.00
```

Note that the ** must be the first spec in the fixspec. Also ** cannot be used along with Z because Z leaves no unused digit positions.

❖ *Note:* If you do not reserve a character position for the sign, and the value in the PRINT(#) USING statement is negative, the sign will be displayed in the rightmost unused character position. If there is no unused position, the entire field will be filled with exclamation points, indicating that the number of digits exceeds the number of places specified for them. Therefore, most numeric specs should include a character that reserves a position for the sign.

You can print the dollar and number signs in the rightmost position by using $$ or ++, or in the leftmost position by using --. For example:

```
)PRINT USING "$$+6#.3#"; 09999
$+9999.00
)PRINT USING "++6#.3#"; 09999
+9999.00
)PRINT USING "$--6#.3#"; 09999
$9999.00+
```

You can also place the sign of the number at the end of the fixspec to have it appear and the end of the output:

```
)PRINT USING "$$6#.3#+"; 09999
$ 9999.00+
```

Because Z prints leading zeros, taking up all unused positions, you cannot use it with $$, ++, **, or --, which shift characters to the right, displacing any spaces. (The spaces remain to the left of the field so that its width does not change.)

The best way to learn about using fixspecs is to experiment with various formats. Here is a program that can help:

```
5 REM NumericSpecTester
10 INPUT "Enter desired spec: "; SPEC$
15 ON ERR PRINT "You entered a bad spec, try again!" : GOTO 10
20 F=1000:PRINT
30 X=1: GOSUB 100
40 X=12: GOSUB 100
50 X=123: GOSUB 100
60 X=1234: GOSUB 100
70 PRINT: PRINT " Spec was: "; SPEC$: PRINT
80 GOTO 10
100 PRINT USING SPEC$; X;: PRINT ,X
110 PRINT USING SPEC$; -X;: PRINT ,-X
120 PRINT USING SPEC$; X/F;: PRINT ,X/F
130 PRINT USING SPEC$; -X/F;: PRINT ,-X/F
200 RETURN
```

The NumericSpecTester program first asks you to enter a spec from the keyboard. It then displays two columns of numbers. The left column lists values displayed according to the spec you entered, while the right column contains the same values output by a PRINT statement using no format statement. You can also use this program to study scispecs and engrspecs, which are described in the next section.

The sScispec: The scientific-specification, called **scispec,** formats numeric output in scientific notation. The scispec is simpler than the fixspec; it has only either one digit or none to the left of the decimal point. The number of digits to the right of the decimal point is defined by # characters, either stated explicitly or by using a repeat factor. The letter E defines the exponent position, and you can use a repeat factor with this character as well. You must allow either three or four character positions for the exponent. For example:

```
)PRINT USING "+#.4#4E"; 3.1415926
+3.1416E+00
)PRINT USING "+.4#4E"; 3.1415926
+.3142E+01
```

When the spec calls for one digit position to the left of the decimal point, the first significant digit of the value is placed there; when there is no digit position to the left of the decimal point, the most significant digit is placed to the right of the decimal point. In either case, the exponent is then calculated to make the displayed value correct.

Notice that with four character positions for the exponent, only two are available for the exponent's digits; with three character positions for the exponent, only one is available for the exponent's digit. If the calculated exponent will not fit in the available space, the entire numeric field is filled with exclamation points.

The letter Z can be used instead of # in a scispec, but the effect is the same. Note that if the sign is not explicitly specified in a scispec, the sign of the value will only be printed if there are enough available character positions and the value is negative.

The engrspec: The engineering specification, called **engrspec,** is closely related to the scispec. It forces the exponent's value to be a multiple of 3, and has a maximum of three digit positions to the left of the decimal point.

You can use either a # to replace leading zeros with spaces or a Z to print leading zeros. For example:

```
)PRINT USING "+3#.4#4E"; 1729
+ 1.7290E+03
)PRINT USING "+3Z.4Z3E"; 1729
+01.729E+3
```

The SCALE function

SCALE is used in conjunction with PRINT (#) USING to shift the decimal point of a displayed value to the left or the right. SCALE uses two arithmetic expressions as arguments. The first argument defines the number of places to the right (or left, if negative) that the decimal point should be moved. The second argument is the actual numeric variable to be output.

SCALE takes ten raised to the power equal to its first argument and multiplies that by the value of its second argument. If the first argument value is 5 and the second is equal to 22435, the value output will be equal to 22435.0E5 in the format specified by the PRINT(#) USING statement. For example:

```
)A&=12345678901234567
)PRINT USING "$$20&##";SCALE(-3,A&)
$12,345,678,901,235        (Note rounding of cents)
```

SCALE enables GS BASIC to handle calculations in cents using long integers, and then to output the results with the decimal point positioned to indicate dollars and cents. You can take the same characters given in the example above and, with a slight change, convert cents to dollars:

```
)A&=12345678901234567
)PRINT USING "$$20&.##";SCALE(-2,A&)
$123,456,789,012,345.67  (Note cents)
```

The first SCALE argument must be in the range of -128 to 127, and the resulting exponent of the value must be between -99 and +99, or the

```
?ILLEGAL QUANTITY ERROR
```

message will be displayed.

Chapter 4

Controlling Program Execution

Assignmebt statements xx

The reserved word LET xx
The reserved word SWAP xx

Remark statements xx

The reserved word REM xx

Branching xx

Unconditional branching xx
    The GOTO statement xx
Conditional branching xx
    IF ... THEN statements xx
    ELSE clauses xx
    Multiline IF ... THEN ...ELSE statements xx
    Nested conditional statements xx
    Conditional statement considerations xx

Looping xx

FOR ... NEXT statements xx
STEP clauses xx
Do ... WHILE ... UNTIL statements xx
    The reserved verb UNTIL xx
    The reserved verb WHILE xx
    The reserved verb DO xx

Subroutines xx

GOSUB statements xx
RETURN statements xx

POP statements xx

Computed branching xx

ON ... GOTO statements xx
ON ... GOSUB statements xx
ON KBD and OFF KBD statement xx
The reserved variable KBD xx

Handling errors xx

ON ERR and OFF ERR statements xx
RESUME statements xx
The reserved variables ERR and ERRLIN xx

Error-recovery strategies xx

This chapter describes the statements and functions supplied by Apple IIGS BASIC to help you control the path of execution of your programs. Even a short program will use one or more of the items described here, and a large program might use all of them.

Assignment statements

You use **assignment statements** to assign the values of expressions to variables. The variable to the left of the equal sign is assigned the value of the expression on the right side. For example:

```
)Ripple=5*4^1/2
)PRINT  Ripple
10
)
```

The following are examples of assignment statements:

```
)IV%=9+PMT
)DSJ&=234324
)10  Body$(Arm,Leg)="Bone"
)10  ReassuringWords=Comforting-12
```

You can use variables and expressions of any type in an assignment statement. However, if the type of the variable on the left side of the replacement sign is different from the one on the right side, type conversion must occur. BASIC automatically converts integer and real variables and numeric expressions, but you must handle string-numeric cases explicitly. A string expression may not be assigned directly to a numeric variable or vice versa. (See the section titled "Functions" in Chapter 2 for an explanation of type conversion functions.)

The reserved word LET

LET may optionally precede an assignment statement. For example:

```
)LET  Henry=FatherofJack
```

Although the reserved word LET is not required, it can make a program listing somewhat easier to read and understand. If the variable on the left side of the equal sign is a string variable, the expression on the right must be a string expression. If the variable is a numeric variable, the expression must be a numeric or logical expression. Otherwise, you will see the message

```
?TYPE MISMATCH ERROR
```

❖ *Note:* You can make only one assignment per statement. For example, the statement

```
)A=B=0
```

does not assign the value 0 to both A and B; instead, BASIC evaluates the logical expression B=0 and assigns the result to variable A. (See the discussion of logical expressions in Chapter 2.)

The reserved word SWAP

SWAP exchanges the value stored in one variable for the value stored in another. The names of the two variables whose values are to be swapped must follow the reserved word SWAP. For example, the statements

```
)A=4 : B=8 : C=A/B
```

store the value .5 in variable C. But

```
)A=4 : B=8 : SWAP A,B : C=A/B
```

stores the value 2 in variable C.

You can use string, integer, double-integer, long-integer, and single- or double-real variables with SWAP, but both of the variables to be exchanged must be of the same type. If the two variables are not the same type, a

```
?TYPE MISMATCH ERROR
```

message appears.

Remark statements

Since programs are not written in natural languages such as English, they are not always easy to understand. Remarks clarify the purpose and methods of your programs. Use them generously to allow other programmers to maintain your programs, as well as to remind yourself what your programs are supposed to do and how they do it.

The reserved word REM

REM allows you to insert remarks into your program. BASIC ignores everything in a program line following the reserved word REM, but carries this text along with the rest of the program. For example, if you type:

```
REM Munge BASIC : Bake Apple : Inverse : PRINT
```

BASIC will not execute any of the words or statements following the reserved word REM. You can tell that this is so by entering reserved words in lowercase; executable reserved words are displayed by the BASIC LIST command in uppercase.

The reserved word REM must be the first thing in a statement, or BASIC will not treat it as a remark. For example:

```
)HPOS REM arkably Tall Buildings
```

is not a legal statement; however, the statement:

```
)REM HPOS kyscraper
```

is leagal.

Like all other statements, REM statements must not exceed 239 characters in length. If you comment your programs heavily, use several REM statements in successive lines.

Branching

A program is said to **branch** when it does not execute the next higher numbered statement in sequence but, *jumps* to some other line instead. There are two kinds of branching: conditional and unconditional.

Branching in GS BASIC can be by reference to line numbers, and programs can get quite confusing if you are not careful. Because line numbers alone are not very meaningful, BASIC allows an optional label to be included on any line in your programs, and those labels can be used in place of line numbers in branching statements. It is good programming practice to give only the beginning line of a related group of statements a label—one that is descriptive of the function of the entire group.

Unconditional branching

A statement causing program execution to branch each time it is executed, under all conditions, is known as an **unconditional branch.**

The GOTO statement

GOTO causes execution of the program to jump to the beginning of a specified statement list. You specify the statement list to which execution should jump by following the reserved word GOTO with the line number or label of the statement list. For example:

```
)10 PRINT 10 : GOTO PGMEND
)20 PRINT 20
)30 PRINT 30 : STOP
)40 PGMEND: PRINT 40
)50 PRINT 50 : GOTO 20
)RUN
10
40
50
20
30

PROGRAM INTERRUPTED IN 30
)
```

Most versions of BASIC begin at the lowest line number and search sequentially until they find the desired line. Apple IIGS BASIC uses a more expedient method. If the line number referenced by the GOTO statement is greater than the number of the GOTO, it begins searching at the current line; otherwise, the search begins at the start of the program. However, a if a label is used, Apple IIGS BASIC always searches the entire program.

If the line number given in the GOTO statement does not exist, or if there is no line number given, you will see an error message. For example, the message

```
?UNDEF'D STATEMENT ERROR IN 5293
```

means that 5293 is the line number of an erroneous GOTO statement. Immediate execution of a similarly illegal GOTO statement such as

```
)GOTO 45
```

when there is no program in memory will generate an

```
?UNDEF'D STATEMENT ERROR
```

message.

Conditional branching

A statement causing program excecution to branch only under certain conditions is called **conditional branching.**

IF ... THEN statements

IF statements allow the order of execution of statements to depend on the truth value of a logical expression. The IF statement must include both a logical expression to be evaluated and instructions for BASIC to follow if the expression is true. If the expression is false, execution passes to the next program line beyond the entire IF statement in the program, and BASIC ignores the instructions given in the IF statement.

The logical expression to be evaluated must follow the reserved word IF, and the instructions must follow the reserved words THEN. BASIC also allows the verb GOTO to be used in place of the words THEN, but GOTO must be followed by a line number or a label; unlike THEN, GOTO may not be followed by a statement.

BASIC also allows you to follow THEN with a line number or a label in addition to a statement list. For example:

```
)10530 IF A=4 THEN PRINT.REPORT
) 3700 IF KP+BH GOTO 3785
) 100 IF G& MOD F& >2 GOTO 121
```

In an IF ... THEN statement, the THEN or GOTO can be followed by any line number to which execution should branch or a statement list for BASIC to execute. For example:

```
)IF 0 THEN PRINT 1
)50 IF 2+2 THEN 2500
)IF S/4>=17 * NOT 2 THEN GOSUB 3000 : INVERSE : PRINT "Hi"
)IF Language=German THEN PRINT "Gesundheit" :
Sneezes = Sneezes + 1
)10 IF Language=English THEN PRINT "Bless you!" :
Sneezes = Sneezes + 1
```

These are all equivalent statements:

```
)IF G=5 THEN 200
)IF G=5 GOTO 200
)IF G=5 THEN GOTO 200
```

ELSE clauses

The ELSE clause of the IF statement allows you to specify instructions for BASIC to execute if the truth value of the logical expression is false. In other words, when the expression is false, instead of having execution pass to the next line after the IF statement, you can have BASIC execute some other instructions.

The instructions following the reserved word ELSE can be a line number to which execution should branch or a statement list to execute. If the logical expression is true, BASIC will skip the ELSE clause and any statements following on that line. For example:

```
)IF X=1 THEN Y=2 : ELSE Y=3
)IF 3<PL5 THEN PL5=-PL5 : ELSE NORMAL : GOTO 376
)718 TEXT : IF NOT Y THEN 3200 : ELSE TEXTPORT 1,1 TO 4,4
 : GOTO 457
```

and

```
)10 IF 0<5 THEN PRINT 101 : ELSE PRINT 100
)20 IF 0>5 THEN PRINT 201 : ELSE PRINT 200
)RUN
101
200
```

BASIC treats an ELSE clause that does not immediately follow an IF ... THEN statement as if it were a REM statement. For example, BASIC would ignore the line

```
)ELSE Whatever you want to remark about here!
```

Multiline IF ... THEN ... ELSE statements

IF statements with long statement lists in either the THEN or ELSE clauses may not fit on a single program line. In GS BASIC you can separate an IF statement into two or three program lines by breaking it at the THEN or ELSE verbs. For example, the statement

```
100 IF RTYP=7 THEN PRINT "NAME:";RNAME$
110 ELSE PRINT "WRONG RECORD TYPE"
```

or

```
200 IF RTYP=7
210 THEN PRINT "NAME:";RNAME$
220 ELSE PRINT "WRONG RECORD TYPE"
```

will execute as if it had been written on a single line.

The IF statement can be continued on the next line of a program only if the continuation lines begin with the verb THEN or ELSE. Furthermore, any line beginning with THEN or ELSE that directly follows an IF statement is always treated as part of that statement. Note that the continuation of an IF statement only occurs at line boundaries, not at statement boundaries within a program line.

The next section discusses IF statement nesting, which can be done within a multiline IF statement. Remember that the line continuation must always occur at the THEN or ELSE verb of any IF statement, no matter how deeply nested.

By combining multiline IF and nested IF statements, you can build an IF statement that spans many program lines. Although it may appear that you can branch into the middle of an IF statement, you cannot do this; since, the THEN and ELSE clauses act exactly like REM statements unless they are executed by an immediately preceding IF statement.

Nested conditional statements

The statement list following a THEN or ELSE clause or continuation line can contain as many additional IF ... THEN or IF ... THEN ... ELSE statements as the 239-character limit will allow. Conditional statements contained inside other conditional statements in this manner are said to be **nested.** In these cases, BASIC matches each ELSE with the most recently encountered and unmatched THEN. For example:

```
)IF INNING=9 AND TEAM=HOME THEN IF MEN.ON.BASE=3 THEN PRINT "bunt"
: ELSE PRINT "swing at it!!"
```

In this example, the ELSE clause goes with the second THEN, which is the one most recently unmatched. As another example:

```
)IF toothache THEN IF dentist is on vacation THEN suffer
: ELSE call for appointment : ELSE smile!
```

This example could be organized into multiple lines, as follows:

```
 1000 IF toothache
 1001 THEN IF dentist is on vacation
 1002 THEN suffer
 1003 ELSE call for appointment
 1004 ELSE smile!
```

Conditional statement considerations

The following IF statements have legal logical expressions:

```
)IF 1<2 THEN PRINT "Yup"
)IF 0 THEN PRINT "Nope"
)IF A<B THEN PRINT "Yup" : ELSE PRINT "Nope"
)IF "A"<"B" THEN PRINT "Yup"
)IF 1+2*Z THEN PRINT "Maybe so" : ELSE PRINT "Maybe not"
```

Unlike in most other versions of BASIC, the following IF statements are also legal in Apple IIGS BASIC:

```
)IF "Fred" THEN PRINT "Fred"
)IF KAREN$ THEN PRINT "Fred's friend"
)IF "Fred"+KAREN$ THEN PRINT "Fred 'n Karen"
```

Normally, a logical expression can be either a comparison of arithmetic expressions or a comparison of string expressions. It is important to remember that logical expressions reduce to a truth value of true or false, represented by an integer value of 1 and 0, respectively.

It is legal to substitute a single arithmetic expression for a logical expression. However, you can only substitute a single string expression for a logical expression in the conditional expression of an IF statement.

All BASICs standardly treat the value of an arithmetic expression as true or false or nonzero or zero. GS BASIC has similar rules for string expressions. It recognizes a string result and uses the length of the string, a number from 0 through 255, as the logical result. A length of zero has a truth value of false and a nonzero length has a truth value of true.

Thus, IF **string** THEN is an intuitive language extension, and it functions the same as if you used the LEN function (described in the "String Functions" section of Chapter 2). For example:

```
)IF LEN(KAREN$) THEN PRINT "Karen exists!"
: ELSE PRINT "No Karen"
```

An IF without a matching THEN or GOTO generates a

```
?SYNTAX ERROR
```

message.

ELSE must be preceded by a colon when it follows a THEN clause on the same program line. For example:

```
100 IF X=12 THEN 1000 : ELSE 2000
```

However, ELSE need not be preceded by a colon when it begins a separate line in a multiline IF ... THEN ... ELSE statement, as follows:

```
200 IF Minnow$=Fish$ THEN Answer=True
210 ELSE Answer=False
```

Looping

Looping is the process of carring out one or more operations repetitively. Loops can be divided into two general types: those operating a determined number of times and those operating either *as long as* a specified condition is true or *until* a specified condition is true. The first type uses the FOR ... NEXT structure described below, and the second type is accomplished with the WHILE ... UNTIL or the DO ... UNTIL loop statements, described later in this chapter.

FOR ... NEXT statements

The verbs FOR and NEXT allow a group of statements to be executed a specified number of times. The FOR statement defines the beginning of the statement list making up the body of the loop and sets the number of times it is to be executed, and the NEXT statement defines its end. For example, if you wanted to display the numbers 1 to 5, you might write a program like this:

```
)10 Number=1
)20 PRINT Number
)30 Number=Number+1 : IF Number<6 THEN 20
```

Alternatively, you could use the FOR and NEXT statements like this:

```
)10 FOR Number=1 TO 5
)20 PRINT Number
)30 NEXT Number
)RUN
1
2
3
4
5
)
```

In line 10, the **control variable** Number is assigned a beginning value of 1 and an ending value of 5. The NEXT statement in line 30 of this program functions the same as the statements in line 30 of the previous program. It increments the value of Number by 1, and then checks to see if the value of Number is greater than the ending value that was specified in the FOR statement.

If the value of Number is less than the ending value, execution loops back to the statement immediately following the FOR statement (line 20). If the value of the control variable is greater than the ending value, execution continues with the next statement immediately following the NEXT statement.

The control variable can be any type of numeric variable, either real or integer, but it cannot be a string variable. The beginning and ending values assigned to the control variable in the FOR statement can be the result of arithmetic expressions. For example:

```
)FOR Repeat%=T+44-F6/D*NOT R TO 54*5/FJ : NEXT Repeat%
```

is perfectly legal, if somewhat obscure.

BASIC supports five types of numeric variables, single- and double-precision reals, and regular, double, and long integers. The initial value, limit value, and optional increment value of a FOR loop are all converted to the type of the control variable when the FOR statement is executed. The conversions are performed as if the FOR statement were written like this:

```
10 FOR I% = CONV%(init) TO CONV%( limit) STEP CONV%(increment)
```

A similar statement could be written if the control variable were I, I#, I@, or I& by substituting the functions CONV, CONV#, CONV@, and CONV&, respectively. These forced conversions (type coercion) allow maximum speed during the NEXT addition and compare operations.

Due to these coercions, a FOR loop with a regular integer control variable will execute up to six times faster than a real or long-integer control variable. Double-integer loops will operate up to three times faster. This design allows for speed optimization, but it creates a restriction.

A FOR loop with an integer control variable cannot have a nonintegral initial, limit, or increment value. If one is used, it will be rounded to the nearest whole number by the FOR statement, and the rounded result will replace the value given during loop execution. This rounding is done without any warning or error message.

FOR ... NEXT loops may contain other FOR ... NEXT loops; loops contained within other loops are said to be nested. For example:

```
)10 FOR Row%=1 TO 3
)20 FOR Column=2 TO 3
)30 PRINT Row, Column
)40 NEXT Column
)50 NEXT Row%
```

Nested FOR ... NEXT loops will align with the other statements when you enter them (as shown above), but they will be indented when displayed by LIST. For example:

```
)LIST
 10 FOR Row=1 TO 3
 20   FOR Column=2 TO 3
 30     PRINT Row, Column
 40   NEXT Column
 50 NEXT Row
```

Notice that the NEXT statements are lined up underneath the matching FOR statement, the inner loop is indented, and the body of the inner loop is indented again. The number of characters for each indentation level is controlled by the modifiable reserved variable INDENT, as described in Chapter 1, in the section titled "The reserved variables INDENT and OUTREC."

NEXT statements can contain as many variable names as you would like. For example:

```
)60 FOR Loop1=1 TO 4 : FOR Loop2=4 TO 55
)70 FOR Loop3=-4 TO 55
)80 NEXT Loop3, Loop2, Loop1
```

The first control variable given in the NEXT statement must be the same as the one named in the most recently executed FOR statement; the second control variable given must match the second most recently executed FOR statement, and so on. Incorrectly matched FOR and NEXT statements cause the message

```
?NEXT WITHOUT FOR ERROR
```

to be displayed. The following example contains incorrectly nested loops:

```
)90 FOR A=1 TO T%
)100 FOR B=1 TO 43
)110 NEXT A, B
```

The FOR statement scans forward in the program, searching for the matching NEXT statement before the body of the FOR loop is executed. If the NEXT statement is not found, the message

```
?FOR WITHOUT NEXT ERROR
```

will appear. This look-ahead scan properly accounts for nested FOR ... NEXT loops when locating the matching NEXT statement, but it only considers NEXT verbs that begin a statement. A NEXT verb embedded in the THEN or ELSE clause of an IF statement is ignored. This allows a FOR statement to have more than one NEXT statement. Some additional details on the look-ahead matching are described in Chapter 8, "BASIC Reference."

STEP clauses

FOR statements may optionally include a STEP clause, allowing you to specify the amount to increment the control variable with each iteration of the loop. For example:

```
)FOR BYTE=2 TO 10 STEP 3 : PRINT BYTE : NEXT BYTE
2
5
8
)
```

If you do not use a STEP clause, the control variable is incremented by 1, by default, when the following NEXT statement is executed. You can use any arithmetic expression to specify the value to increment the control variable. If the control variable is any type of integer, the value of the arithmetic expression will be rounded to the nearest integral value by the implied type conversion.

If you specify a negative value in a STEP clause, the loop counts backwards. For example:

```
)FOR Loop=10 TO 1 STEP -2 : PRINT Loop : NEXT Loop
10
8
6
4
2
)
```

If there is a negative increment value in a STEP clause, and the value of the control variable is less than the ending value after it has been incremented by a NEXT statement, execution passes to the statement following the NEXT; in other words, the loop is terminated when the NEXT statement is reached. For example:

```
)FOR Loop=1 TO 10 STEP -2 : PRINT Loop : NEXT Loop
1
)
```

If the increment value is 0, the control variable is incremented by 0 each time a NEXT statement is executed. This means that the value of the control variable will never be greater than the ending value, and the statements between FOR and NEXT will be repeated indefinitely (unless the ending value is less than the control variable at the start). This is known as an infinite loop. The only way to stop the looping is to press Control-C.

- *Note:* The control variable is incremented and compared to the ending value only when the NEXT statement at the end of the FOR ... NEXT loop is executed. This means that the statements between the FOR and NEXT are *always* executed at least once.

A NEXT statement without a specified control variable defaults to the control variable given in the most recently executed FOR statement still in effect, and it executes faster than one with a specified control variable. For example:

```
)FOR G6=4 TO REV3/21 : NEXT
```

If there is no FOR ... NEXT loop in effect, executing a NEXT statement generates a

```
?NEXT WITHOUT FOR ERROR
```

message. The same message is displayed if the control variable specified by a NEXT statement is different than the control variable given in the most recently executed FOR statement.

Nesting more than nine FOR ... NEXT loops inside one another results in the message

```
?STACK OVERFLOW ERROR
```

If a deferred execution FOR statement is still in effect, an immediate execution NEXT statement can cause a jump to the deferred execution program, where appropriate.

DO ... WHILE ... UNTIL statements

The verbs DO, WHILE, and UNTIL can be combined in six useful combinations to create conditional looping logic similar to FOR ... NEXT loops. WHILE ... UNTIL loops do not have explicit control variables like FOR loops, but they do have a body of statements that can be executed repeatedly. An example of each general type is show below:

```
  100 DO : statements … : UNTIL GH > 29
14000 WHILE GH >= 299: statements … :UNTIL
  550 DO : statements : WHILE twgy > 11 : statements : UNTIL
```

The type in the first example is called a DO ... UNTIL loop, the type in the second example is called a WHILE ... UNTIL loop, and the last example is called a DO ... WHILE ... UNTIL loop.

The reserved verb UNTIL

UNTIL is used in conjunction with DO and/or WHILE to create various types of conditional loops. The verb UNTIL marks the end of the loop construct and can be used with or without a logical expression.

If you use UNTIL without a logical expression, it loops back to the most recently executed DO or WHILE statement (DO takes precedence if both a DO and a WHILE precede the UNTIL). When you use a logical expression, UNTIL loops back if the expression is false (zero) and proceeds to the next statement if the expression is true (nonzero). Thus, a DO ... UNTIL loop always executes the body once and continues to loop until the condition becomes true. If the condition is true when the DO is first executed, the body of the loop will be executed only once.

You can construct an infinite loop with a DO ... UNTIL statement by omitting the logical expression following the reserved word UNTIL. DO and UNTIL need not be on the same program line. A DO ... UNTIL loop displayed by the LIST command is indented, like a FOR ... NEXT loop.

You can easily duplicate the WHILE ... WEND or REPEAT ... UNTIL constructs implemented by other versions of BASIC with WHILE ... UNTIL statements by simply including or excluding the logical expression. Furthermore, you can use the WHILE ... UNTIL construct in new combinations. The following common combinations are possible:

Apple IIGS BASIC loop construct	Common name
`WHILE lexpr : … statements … : UNTIL lexpr`	WHILE ... UNTIL
`WHILE lexpr : … statements … : UNTIL`	WHILE ... WEND
`WHILE : … statements … : UNTIL lexpr`	REPEAT ... UNTIL
`DO : … statements … : UNTIL lexpr`	DO ... UNTIL
`WHILE : … statements … : UNTIL`	infinite loop

In addition, there are two DO ... WHILE ... UNTIL constructs:

```
DO : statements : WHILE lexpr :… statements …:UNTIL
DO : statements : WHILE lexpr :… statements …:UNTIL lexpr
```

In a DO ... WHILE ... UNTIL loop, BASIC *always* executes the statements before the WHILE and *conditionally* executes the statements after the WHILE. This effectively separates the body of the loop into two parts; the prefix part is unconditionally executed at least once, and the conditional part may be executed zero, one, or more times.

UNTIL examines the control stack for the WHILE information. If it cannot find a prior DO or WHILE statement, the message

```
?UNTIL w/o WHILE ERROR
```

appears.

The reserved verb WHILE

The verb WHILE marks the beginning or midpoint of a loop construct and can be used with or without a logical expression. Using WHILE between DO and UNTIL without a logical expression is a meaningless (although valid) construct.

If you use WHILE without a logical expression, it behaves as if a true expression were present. When you use a logical expression, WHILE executes the following statements if the expression is true (nonzero), and skips to the statement following the UNTIL if the expression is false (zero). The logical expression in the matching UNTIL statement does not influence the behavior of WHILE.

WHILE searches forward in the program for a matching UNTIL and will display the message

```
WHILE w/o UNTIL ERROR
```

if the UNTIL is not present.

The reserved verb DO

The DO statement defines the beginning of a DO ... UNTIL loop or a DO ... WHILE ... UNTIL loop. The DO statement does not look ahead for the UNTIL statement, but an intervening WHILE statement will search for a matching UNTIL statement.

Since DO doesn't look ahead, a DO ... UNTIL construct may have multiple UNTIL statements, some of which are conditionally executed within an IF statement. A DO ... UNTIL loop with multiple UNTIL statements must be carefully coded so that one conditional UNTIL statement doesn't lead to another unconditional UNTIL statement. When this occurs, the message

```
?UNTIL w/o WHILE ERROR
```

appears.

Subroutines

A subroutine is a group of statements that perform some specialized or frequently repeated task. BASIC allows you to include a descriptive label on the beginning line of a subroutine so that your programs can reference the subroutine by name instead of by line number. Such labels make your programs much easier to understand.

IIGS BASIC supports another type of subroutine, called a procedure, which is described in Chapter 7, "Advanced Topics."

GOSUB statements

GOSUB causes BASIC to branch to a subroutine. You must follow the reserved word GOSUB with the line number or label of the first statement in the subroutine. When BASIC executes a GOSUB statement, it places a pointer to the statement immediately following that statement it places at the top of a list of pointers, called a **Control stack,** then transfers execution to the line number given in the GOSUB statement.

If the line number or label given in a GOSUB statement does not correspond to an existing line in the program, BASIC displays a message. For example

```
?UNDEF'D STATEMENT ERROR IN 746
```

means that line 746 contains an erroneous GOSUB statement.

RETURN statements

RETURN has no parameters when paired with GOSUB. In executing a RETURN statement, BASIC removes one pointer from the top of the Control stack and branches to the statement indicated by the pointer. This is normally the statement immediately following the most recently executed GOSUB statement. For example, in this program:

```
)10 GOSUB 200
)20 PRINT "Back"
)30 END
)200 REM The subroutine goes here
)210 RETURN
```

line 10 places a pointer to the next statement (line 20) on top of the Control stack, and branches execution to the subroutine at line 200. After line 200, execution transfers to line 210, where the RETURN statement causes BASIC to remove the top pointer from the control stack and branch to the line indicated (line 20).

Here is another example of GOSUB and RETURN statements:

```
)10 PRINT "Now executing subroutine 100"
)20 GOSUB 100
)30 PRINT "Hello again"
)40 END
)100 PRINT "Subroutine 100 speaking"
)110 RETURN : REM Line 30 will now be executed.
```

A program can have nested subroutines (subroutines calling other subroutines) up to about 40 levels deep. If GOSUB statements are nested more than about 40 levels, the

```
?STACK OVERFLOW ERROR
```

message is given. For example, here is a program with subroutines nested three levels deep:

```
)10 GOSUB 100
)20 END
)50 GOSUB 200
)60 RETURN : REM Branch to 120
)100 GOSUB 50
)120 RETURN : REM Branch to 20
)200 RETURN : REM Branch to 60
```

Note that the BASIC statement RETURN is not the same as pressing the Return key. The RETURN associated with subroutines is a normal BASIC statement, and the word is spelled out.

If BASIC attempts to execute one more RETURN statement than it has encountered GOSUB statements, the

```
?RETURN WITHOUT GOSUB ERROR
```

message is displayed.

POP statements

POP allows you to jump out of one level of subroutine nesting.

When a POP statement is executed, BASIC removes (pops) the top pointer from the program stack and discards it, without causing execution to branch anywhere. When BASIC encounters the next RETURN statement after a POP statement is executed, instead of branching to the first statement beyond the most recently executed GOSUB, it branches to the first statement beyond the second most recently executed GOSUB.

If a POP statement is executed before a GOSUB has been encountered, or if more POP statements and RETURN statements are encountered than GOSUB statements, BASIC displays the message

```
?RETURN WITHOUT GOSUB ERROR
```

because more pointers have been removed from the stack than were placed on it.

Here is an example of the use of POP:

```
)NEW
)10 GOSUB 100
)20 PRINT "End of program"
)30 END
)100 REM This subroutine has no RETURN statement
)110 PRINT "Subroutine 100 speaking"
)120 PRINT "About to branch to subroutine 200"
)130 GOSUB 200
)140 REM This line is never executed
)200 PRINT "Subroutine 200 speaking"
)210 PRINT "Popping to avoid returning to line 140"
)220 POP : REM Removes pointer to line 140 from stack
)230 RETURN : REM Execution now resumes at line 20
)RUN
Subroutine 100 speaking
About to execute subroutine 200
Subroutine 200 speaking
Popping to avoid returning to line 140
End of program
)
```

Computed branching

Many programs require a different set of operations for each possible value in a range. **Computed branching** allows you to tailor your program to respond to a number of possible conditions.

ON ... GOTO statements

ON ... GOTO statements are used to specify different program branch points, based on the value of an arithmetic expression. The arithmetic expression must follow the reserved word ON, and the line numbers or labels to which execution should branch must follow the reserved word GOTO. For example:

```
)1000 ON X GOTO 100, 10, 300, 40, PRINT.REPORT
```

If X=1, execution branches to the first line in the list of numbers (line 100); if X=2, execution branches to the second line in the list (line 10); if X=3, execution branches to line 300 (the third line in the list); and so on. Remember that you can use a label in place of a line number to make your programs more readable.

The value of the arithmetic expression must be within the range of 0 through 255, or you will see the message

```
?ILLEGAL QUANTITY ERROR
```

If the value of the arithmetic expression is 0 or greater than the number of line numbers or labels given in the ON ... GOTO statement, BASIC ignores the list of line numbers, and execution continues with the next statement in the program.

ON ... GOSUB statements

ON ... GOSUB statements are identical to ON ... GOTO statements, except that the line numbers or labels following the reserved word GOSUB must reference subroutine entry points. For example:

```
)1000 ON X GOSUB 1000, 2000, 3000, INIT.DISK
```

When BASIC executes a RETURN statement within the subroutine, execution branches to the statement immediately following the ON ... GOSUB statement.

As with ON ... GOTO statements, the value of the arithmetic expression must be within the range of 0 through 255, or you will see the message

```
?ILLEGAL QUANTITY ERROR
```

If the value of the arithmetic expression in the ON...GOSUB statement is 0, or greater than the number of line numbers or labels given, BASIC ignores the list of line numbers, and execution continues with the next statement in the program.

ON KBD and OFF KBD statements

ON KBD causes BASIC to execute a specific statement list immediately when any key is pressed. The statement list to be executed must follow the reserved word KBD.

After an ON KBD statement has been executed, BASIC continues executing the program normally—but as soon as any key is pressed, execution branches back to the most recently executed ON KBD statement. Then the statement list pointed to by the ON KBD statement is executed.

BASIC treats the branch to the ON KBD statement list as a GOSUB statement branch to a subroutine, so the program segment that ON KBD causes to be executed must end with a RETURN statement. To enable ON KBD to handle more than one keystroke, the last statement in the list should be another ON KBD statement. For example:

```
10 ON KBD GOTO 100 : REM 'GOSUB 10' when any key is pressed
20 PRINT "."; : REM Print periods while not handling key-strokes
30 GOTO 20
100 PRINT KBD : REM Display the ASCII value of the key pressed
110 ON KBD GOTO 100 : REM Reenable ON KBD before RETURN
120 RETURN : REM Program continues executing wherever it was
```

This program displays many periods, and whenever a key is pressed, BASIC executes the instructions in the ON KBD statement.

BASIC forgets the last ON KBD statement as soon as a key is pressed, even before it executes the statement list in the ON KBD statement. This is why the program above includes another ON KBD statement in line 110. BASIC also forgets the last ON KBD statement executed if the program returns to immediate execution.

Execution of an OFF KBD statement causes BASIC to forget the last ON KBD statement that was executed.

An ON KBD statement must be executed just prior to the RETURN statement, or a

```
?STACK OVERFLOW ERROR
```

message may appear.

❖ *Note:* When ON KBD is in effect, you cannot halt the program by pressing Control-C because it is treated like any other keystroke. However, the ON KBD statement could cause a branch to a STOP or END statement if a control-C is pressed.

The reserved variable KBD

Apple IIGS BASIC allows you to read, but not write to, its reserved variables. KBD contains the ASCII value of the last key pressed (see Appendix A, "ASCII Character Codes"). When you use the reserved variable KBD in an ON ... GOTO or ON ... GOSUB statement, you must enclose KBD in parentheses, so that BASIC will not confuse it with an ON KBD statement. For example:

```
)ON (KBD)-64 GOTO 100,200,300
```

Handling errors

Apple IIGS BASIC provides several tools that allow your programs to handle anticipated errors. These are especially useful if other people will be using your programs.

ON ERR and OFF ERR statements

ON ERR is used to force BASIC to let your program handle any errors that might occur. When an ON ERR statement is not in effect and an error occurs during deferred execution, BASIC displays an error message on the screen and halts execution. ON ERR is typically used to give a more informative error message or to provide the user with a chance to avoid causing another error. The ON ERR statement should not be used as a tool for finding errors in programs. (Use the TRACE statement, described in Chapter 1 for this.)

A statement or statement list must follow the reserved variable ERR. Execution branches to the subroutine referenced by the statement list whenever an error is encountered; BASIC does not display its error message or halt execution.

For the ON ERR statement to be most effective, it should appear near the beginning of the program (BASIC must execute it before it can use it). Errors that occur before an ON ERR statement result in BASIC's normal error responses.

If a program contains more than one ON ERR statement, only the most recently executed one will be used.

OFF ERR cancels the most recently executed ON ERR statement. There are no parameters or options associated with this statement. After an OFF ERR statement has been executed, BASIC resumes displaying error messages and halting execution in response to an error, just as it did before the ON ERR statement was executed.

Warning:

The statements that ON ERR causes to be executed must themselves be free of errors, or an endless loop may result. You can halt the endless loop by pressing Control-C. (Control-C is handled separately by the ON BREAK statement.) For a complete list of BASIC errors, see Appendix B, "Error."

The following program illustrates one simple way to use the ON ERR statement. In this example, the computer is expecting the user to enter a number. The error-handling statements are executed if a letter or word is typed instead.

```
10 REM EXAMPLE OF ERROR HANDLING
20 ON ERR GOSUB 1000
30 INPUT "Please type a single number between 1 and 100 ";X
40 PRINT "The number you typed was ";X
50 END
1000 REM ERROR HANDLING SUBROUTINE
1010 PRINT : PRINT "I'm very sorry, but only a number will do. Please try again."
1020 RETURN
```

RESUME statements

If your error-handling routine ends with a RESUME statement, execution will begin again at the statement where the original error occurred. In addition to RESUME with no options, you can execute RESUME NEXT in your error-handling routines.

RESUME NEXT skips the statement that caused the error and returns control to the next statement within the program. Using RESUME NEXT requires care to ensure that the program will function in a useful manner when a given statement is ignored. You can control this behavior by only using RESUME NEXT for specific cases in your program and checking for them by examining the reserved variable ERRLIN, described below.

Apple IIGS BASIC ignores any RESUME statements that it encounters until an error occurs. If you try to use RESUME in immediate execution, an

```
?ILLEGAL DIRECT ERROR
```

message appears.

Warning:

ON ERR subroutines using RESUME must be error-free. Errors in your error routines may lock up your system. If this happens, you will have to reboot BASIC, and anything in memory will be lost.

The reserved variables ERR and ERRLIN

When BASIC encounters an error, it assigns the reserved variable ERR a code number corresponding to the type of the detected error. You can then refer to the reserved variable ERR to determine what kind of error occurred. For a list of these codes and the corresponding error messages, see Appendix B "Error messages."

Your error-handling routines called by an ON ERR statement can check the reserved variable ERRLIN to determine which line contained the error.

❖ *Note:* Because multiple statements may appear on a single line, ERRLIN does not determine exactly which statement caused an error. It is advisable to place only one statement per line where errors are most likely so that you can see exactly which statement caused the error.

Error-recovery strategies

You can ask an ON ERR statement to execute any legal BASIC statement. For example, you can use the reserved variable ERR in an ON ... GOSUB statement to handle the individual errors that can occur. Each subroutine could handle the particular error conditions in the most appropriate manner. When the subroutine returns, the last statement in the ERR statement list could be a RESUME statement.

Unfortunately, the concept of restarting statements (using RESUME) in a program is often not a very practical approach to error handling. For example, if the user has entered a filename that doesn't exist, using the RESUME statement to return to an OPEN statement, after issuing a

```
FILE NOT FOUND ERROR
```

message, will just cause the same error to occur again.

A large program will probably need many ON ERR statements for different circumstances; there is seldom one generalized approach that can work effectively in all error-handling contexts. You should use ON ERR whenever you expect an error during a specific activity within a program and treat unexpected errors elsewhere in a program as major problems that require the program to restart everything from a well-defined beginning point.

When an unexpected error occurs, you should attempt to preserve any user data that you can by deleting half complete changes to linked tables or files and closing any open disk files to ensure that the data in disk buffers is written to the media.

Chapter 5

File Handling

Filenames xx

The PREFIX command xx
The modifiable variable PREFIX$ xx

Creating files xx

CREATE statements xx

Manipulating files xx

The CATALOG command xx
DELETE statements xx
RENAME statements xx
LOCK and UNLOCK statements xx

File types xx

Opening and closing files xx

OPEN statements xx
    The FOR options xx
    The FILTYP= option xx
CLOSE statements xx

Accessing text and character files xx

The INPUT# statement xx
The OUTPUT statement xx
The PRINT# statement xx

Accessing data files xx

The READ# statement xx
The WRITE# statement xx

Sequential and random access xx

Sequential access xx
    Sequential access considerations xx
Random access xx
    Random access considerations xx

File statements and functions xx

The ON EOF# statement xx
The OFF EOF# statement xx
The reserved variable EOF xx
The EOFMARK function xx
The FILE function xx
The FILTYP function xx
The TYP function xx
The REC function xx

An example of a file I/O xx

This chapter explains how to use devices and files with Apple IIGS BASIC. You should read the chapter in your *Apple IIGS Owner's Guide* dealing with files to understand files in general and the terminology describing ProDOS files, volumes, and pathnames.

Apple IIGS BASIC treats each peripheral device connected to your computer as a file, including the keyboard, screen, printer, and disk drives. This means that there need be only one method of doing input or output for all the different peripherals connected to your computer. Files on disk volumes are referred to as **disk files,** and files on nondisk devices are referred to as **character files.**

BASIC is designed to support a total of 32 character or disk files. Three of these files are dedicated to fixed purposes: file #0 is used for the .CONSOLE file (consisting of the keyboard and the 80-column text screen display), file #30 is used by the EXEC statement, and file #31 is used by the CAT, CATALOG, and DIR statements.

ProDOS 16 version 1.0 currently allows you to have a maximum of six disk files open at one time, plus one EXEC file, for a total of seven disk files. Although ProDOS 16 allows eight files open at one time, BASIC reserves one for the TYPE, CATALOG, INVOKE, and LIBRARY statements. If you open eight disk files, and then try to use one of these statements, you will see the message

```
?INT/FCB/VCB TBL FULL ERROR
```

indicating that the **FCB (file control block)** table is full.

❖ *Note:* Apple IIGS BASIC actually allows up to 29 open disk files. A future version of ProDOS that supports more simultaneously open files will let you take advantage of this capability.

Filenames

To use any given file, you must refer to it by its **local filename** or **pathname**.

Local filenames can be any sequence of 15 or fewer letters (A through Z), digits, or periods, beginning with an alphabetic character. They may not begin with a slash character (/) or contain spaces.

The local filenames of character devices always begin with a period (.). For instance, .PRINTER, and .CONSOLE are the filenames assigned by BASIC to refer to the slot 1 device, and the slot 3 device (the text screen and keyboard), respectively.

Pathnames may be up to 128 characters in length. The *Apple IIGS Owner's Guide* describes pathnames in detail.

❖ *Note:* You can refer to a specific disk drive by using the device names .D1, .D2, .D3, or .D4 as a partial pathname. The name .D1 will always refer to the drive that contained the boot disk, although the boot disk may not still be in that drive.

Diskettes get their volume names and root directories when they are formatted. To format diskettes, you can use the INIT command, described in Chapter 8.

To create BASIC program files, you must use the SAVE statement. To create BASIC text files, data files, or subdirectories, you must use the OPEN or CREATE statements.

The PREFIX command

The PREFIX command allows you to display and set the eight prefixes supported by ProDOS 16. The current settings for all the prefixes can be displayed if you type

```
PREFIX ?
```

The display will look something like this:

```
0 /GSBASIC/
1 /GSBASIC/
2 /GSBASIC/SYSTEM/LIBS
3
4
5
6
7 /GSBASIC
```

You can view the current value of any individual prefix by following the reserved word PREFIX with a parameter of a single digit from 0 through 7; BASIC will then display the that prefix value on the next line of the screen. PREFIX can be used without the digit or the ? options to display the value of prefix 0.

You can also change any of the eight prefixes using the PREFIX command by following the reserved word PREFIX with a digit, a space, and the pathname of the directory to set into the specified prefix.

To set prefix 0 so that you can use partial pathnames to refer to files, assign a pathname to it with the PREFIX command. For example:

```
)PREFIX 0 /Personnel/Communication/Internal
```

After you have set the prefix, you can refer to files by using either partial pathnames or local filenames. If you wish to access another disk and override the prefix, use a full pathname.

❖ *Note:* When you boot Apple Business BASIC, refix 0 is automatically set to the volume name of the diskette that was used to boot the Apple IIGS BASIC interpreter.

Prefix 7 serves a special purpose for BASIC, and its value is changed by the LOAD, SAVE, SAVE AS, RUN, and CHAIN commands.

The modifiable reserved variable PREFIX$

Pathnames starting with any character other than a period, digit, or a backslash (that is, partial pathnames) are interpreted by BASIC as the contents of the modifiable reserved variable PREFIX$ (which is the same as Prefix 0) concatenated with the partial pathname entered.

A pathname beginning with any alphabetic character is a partial pathname that refers to a file defined by the contents of PREFIX$ plus the partial pathname supplied. A pathname beginning with a / is assumed to be the complete pathname of a file. A prefix beginning with a digit is a partial pathname that refers to a file defined by the contents of prefix n plus the partial pathname supplied.

Creating files

CREATE statements

You use CREATE to make root directories, subdirectories, text files, data files, and any other file types. You must specify the file's name and type in the CREATE statement. The reserved word CREATE is followed by the new pathname, a comma, and the reserved word FILTYP= plus a file type reserved word. The following are the file types:

Table 5-1

Reserved word	Type
TEXT, TXT, or SRC	Text file
BDF or DATA	Data file
DIR or CAT	Subdirectoy

For example, to create a text file called APPLEPIE on a diskette whose volume name is PIES, you would type

```
CREATE "/Pies/Applepie", FILTYP= TEXT
```

Remember that you can use any of the eight prefixes maintained by ProDOS by leaving off the initial backslash of the partial pathname. For example,

```
)PREFIX 5 /PIES
)CREATE 5/Applepie, FILTYP= TXT
```

causes the full pathname to be /PIES/APPLEPIE. The statement

```
)CREATE Fruitpies, CAT
```

creates a subdirectory called Fruitpies (using the prefix specified in prefix 0 as the first part of the pathname) that can contain files.

You can specify the size of each record in a file by appending an arithmetic expression to the CREATE argument list. (A **record** is a sequence of bytes storing data or text.) The record size is required only for random-access files (described later in this chapter), and it must be in the range of 1 through 32767 (BDF files must be in the range of 3 through 32767). For example:

```
)CREATE Attache, TEXT, 4096
```

creates a file with the local filename Attache, having records of 4096 bytes each. If you do not include a record size expression, the files record size defaults to 512 bytes. When creating subdirectories, the arithmetic expression is not allowed.

An attempt to create an already existing file generates a

```
?DUPLICATE FILE ERROR
```

message.

As a convenience in immediate execution, you can enter the pathname directly in CREATE and OPEN statements rather than as a string variable or as a literal enclosed by quotation marks. For example, in immediate execution, the statements

```
)CREATE /Foo/Fighter, FILTYP= BDF
```

and

```
)CREATE "/Foo/Fighter", FILTYP= BDF
```

are equivalent. In deferred mode, the quotation marks are required. Not using them results in the message

```
?TYPE MISMATCH ERROR
```

Manipulating files

BASIC provides several statements for directly manipulating files. By using these statements, you can see what files are on a volume, remove unwanted files, rename files, and lock and unlock files.

The CATALOG command

CATALOG displays a listing of a root directory or subdirectory specified by the pathname following the reserved word CATALOG. If the specified pathname is a diskette volume name, the names of all files in the diskette root directory, as well as those of any subdirectories of the root directory, are displayed. For example, to see a catalog of a diskette named APPLE1, enter

```
CATALOG /Apple1
```

If the pathname specified is a diskette subdirectory, the names of all files in that subdirectory are displayed. For example, if APPLEKIND is a subdirectory, the statement

```
)CATALOG /Apple1/Applekind
```

will list the names of all the files that it contains.

If you specify a partial pathname as the CATALOG argument, the prefix stored in prefix 0 is used. If you specify a single digit from 0 through 7, that prefix is used.

CAT is a short version of the CATALOG command. It displays the first 40 columns of the 80-column display generated by CATALOG. This shorter display generally contains the most useful information from the longer display.

The CATALOG and CAT display information includes a three-character file type field labeled TYPE in the title line. BASIC displays more than 60 of these predefined file type descriptors from an internal table. A complete list of file type descriptors (and the associated value of the file type attribute as stored in the directory entry) is presented in Appendix J, "Common File Types." File types are also discussed in detail later in this chapter.

DELETE statements

DELETE statements are used to remove the subdirectory or file specified as its argument. You can remove a subdirectory only if all the files in it have been deleted. If the last file in a root directory is deleted, the empty root directory will still remain. For example, to delete a file named Banana in a root directory named Tree, you would enter

```
DELETE /Tree/Banana
```

A number of errors can occur when improper pathnames are used with the DELETE statement. They are summarized below.

Table 5-2

Message	Cause
?VOLUME NOT FOUND ERROR	Volume name given does not exist.
?PATH NOT FOUND ERROR	Subdirectory does not exist.
?FILE NOT FOUND ERROR	Local file name given does not exist.
?FILE LOCKED ERROR	Subdirectory contains files, or specified file, is locked.
?WRITE PROTECTED ERROR	Diskette is write-protected.
?FILES OPEN ERROR	The requested file is now open.

RENAME statements

RENAME is used to change the names of volumes, subdirectories, and local files. RENAME's argument list is composed of the old pathname, followed by a comma, followed by the new pathname. For example:

```
)RENAME  /Floppy2/Animals/Dogs,  /Floppy2/Animals/Pigs
```

changes the name of the file Dogs in the subdirectory Animals of the diskette with the volume name Floppy2 to Pigs.

If the second pathname specified indicates a file that already exists, the item is not renamed, and the

```
?DUPLICATE PATHNAME ERROR
```

message is displayed.

Remember that using prefixes reduces the length of the pathname you must type.

You cannot use the RENAME statement to create a file or subdirectory, only to rename an existing one. Use the CREATE statement to make new files and root directories.

A local filename or subdirectory may not be changed to another volume name. For example:

```
)RENAME  /Thisdisk/Tweedledee/File1,  /Thatdisk/Tweedledum/File2
```

will cause the message

```
?BAD PATH ERROR
```

LOCK and UNLOCK statements

LOCK prohibits writing to, saving, or deleting the file named as its argument. Locked files are shown with an asterisk to the left of their file type when cataloged. You can lock subdirectories, but not volume names.

You cannot delete, rename, change, or save a locked file until you have unlocked it with the UNLOCK statement. The reserved word UNLOCK must be followed by the file's name.

To protect all the files on a diskette, you can place a write-protect tab over the write-protect cutout on the upper right edge of the diskette.

File types

Your most useful programs are likely to be those that read from or write to files. The two types of files that your programs will be using are text and data files. Text files contain only text in the form of characters and strings of characters. BASIC automatically converts numeric information stored in text files into string form. BASIC also automatically converts a string representing a numeric value to be assigned to a numeric variable when read from a text file into the proper form.

Reading from or writing to a file is referred to as **accessing** the file. A single access operation usually affects only a portion of the data or text within the file being accessed.

The type of a file is determined at the time that the file is created, either by assignment with a CREATE statement or by the FILTYP= option given with an OPEN# statement. You can change a file type by using the FILTYP= option of the RENAME command, as described in Chapter 8.

Opening and closing files

Before you can access a file, you must open it, and you should close it after you are finished with it. BASIC is designed to allow up to 29 open disk files, however ProDOS 16 version 1.2 allows you to have only up to seven files open at the same time.

OPEN statements

OPEN is used to open files for access, and must precede any file I/O statements accessing a given file. The minimum required arguments following OPEN are the file's pathname, a comma, the reserved word AS, and a file reference number. If you are opening a new file, you must also specify the file type using the FILTYP= option (described in this section).

The file reference number is used in all subsequent I/O statements to refer to the file while it is open. If an OPEN statement contains a file reference number that is already in use, BASIC automatically closes the first file with that number. You can use any file reference number from 1 through 29, but you are limited to seven of these for disk files by ProDOS 16 version 1.2.

❖ *Note:* A useful convention for using file reference numbers in BASIC programs might be to assign character (device) files, such as .PRINTER and .MODEM, file references numbers that are the same as their slot number (1 through 7), and to assign open disk files file reference numbers from 10 through 29.

Here are some examples of OPEN statements

```
OPEN Door, AS #22
OPEN "Window", AS#10
OPEN .PRINTER, AS #1
OPEN .MODEM, AS #2
```

As a convenience in immediate execution, you can enter the pathname directly in OPEN statements, rather than as a string variable or a literal enclosed by quotation marks. For example, in immediate execution, the statements

```
)OPEN/Fpp/Fighter, FILTYP=BDF
```

and

```
)OPEN"/Foo/Fighter", FILTYP=BDF
```

are equivalent. In deferred mode, the quotation marks are required. If you do not use them, you will see the message

```
?TYPE MISMATCH ERROR
```

The FOR options

If the comma after the pathname is followed by the reserved words FOR INPUT, the file is opened as a **read-only file,** which cannot be written to. For example:

```
)OPEN DBMS.INDEX, FOR INPUT AS #12
```

If the comma after the pathname is followed by the reserved words FOR OUTPUT, the file is opened as a **write-only file,** which cannot be read from. For example:

```
)OPEN SESSION, FOR OUTPUT AS #10
```

The FOR APPEND option is a variant of FOR OUTPUT, and it is used in sequential access (explained later) to allow PRINT# or WRITE# statements to append new information to the end of an existing file without disturbing any of its data. For example:

```
)2500 OPEN LADDER$, FOR APPEND AS #11
```

If you do not specify a FOR option, the file is opened with the default option of FOR UPDATE. A file open for update can be both read from and written to if the file type (disk or character) supports such access. For example, you cannot read from a printer device, so BASIC automatically opens a write-only file for it.

The FILTYPE= option

The OPEN statement also has a FILTYP= option that allows you to specify the type of file.

The reserved word FILTYP= follows the comma after the OPEN pathname and precedes the FOR option (if there is one). The file type descriptors that can follow FILTYP= are TXT or TEXT, SRC, BDF or DATA, and DIR or CAT.

ProDOS 16 supports more than 60 file types, even though only a few type descriptors are explicitly supported by the FILTYP= option. You can select any of the remaining file types by using a numeric expression after the reserved word FILTYP=. The value of the expression must be in the range of 0 through 255; otherwise, the message

```
?ILLEGAL QUANTITY ERROR
```

will appear.

Important:

When you are opening a new file, you must include the FILTYP= option to tell BASIC what file type to use when it performs the implied CREATE operation. If you attempt to open a nonexistent file without using the FILTYP= option, you will see the message

```
?FILE NOT FOUND ERROR
```

Also, if you use the FILTYP= option when opening an existing file, and that file has a file type different from the one you specified, the message

```
?FILE TYPE ERROR
```

will be displayed.

❖ *Note:* Programmers familiar with Business BASIC on the Apple /// should note that IIGS BASIC does not support typeless open and first operation of disk files. You must decide what type a file will be when it is opened, and you are restricted to using the proper I/O statement for that type (either TEXT or DATA).

This programming convention would be particularily useful if it were followed by all authors of public domain software, since it would make IIGS BASIC programs easier for others to understand.

Apple IIGS BASIC also provides two additional forms of OPEN that support advanced programming techniques. One of these is described in Chapter 8 under "OPEN," and the other is discussed in Chapter 7, in the section "Opening a Window File."

CLOSE statements

Before the end of your programs, you should use a CLOSE statement to close all open files. Also, any files closed during program execution must be reopened before you can access them again. Each time a file is opened, even if it was used earlier in the same program, BASIC treats it as a new file.

The CLOSE verb is used in two ways. The first option is CLOSE, followed by a # and the file reference number. This closes the file with the given file reference number. The file number argument can be a constant an expression with a value in the range of 0 through 30; any other values will generate the message

```
?ILLEGAL QUANTITY ERROR
```

Warning:
Closing file #0, the .CONSOLE file, from immediate mode will disconnect BASIC from the console, and you will have to reboot your system.

CLOSE, without the file number option closes all files (1 through 29) that are open when the statement is executed. (The LOAD, CLEAR, NEW, and RUN statements also close all open files; the CHAIN statement does not close any files.)

The CLOSE statement closes an open window file by issuing the Window Manager's CloseWindow function call.

Accessing text and character files

You can access both text files and character (device) files by using the same set of BASIC I/O statements: INPUT#, PRINT#, and PRINT# USING. You must use a different set of I/O statements to access BASIC data files (FILTYP=BDF), as described later in this chapter.

The INPUT# statement

INPUT# reads a line of text from a specified file into the input buffer and processes the input text using its list of variables. In this context, ***a line*** is any sequence of ASCII characters, up to 255 characters long, terminated by a return character. If INPUT# does not find a return character after reading 255 characters, it appends one to the buffer and processes the input as a line.

The reserved word INPUT# is followed by the file reference number of the file to be read from, a semicolon, and a variable list, with commas separating the variables. Optionally, you can specify a record number to be read into the input buffer by following the file reference number with a comma and the record number (as an arithmetic expression).

Here are some examples of INPUT# statements:

```
)INPUT# 2; Payment%, Grease$
)INPUT# 8, 34; DG(0), DG(2), DG(4)
```

INPUT# automatically performs any necessary string to numeric type conversions (similar to the VAL function described in Chapter 2) to store newly read information into the numeric variables in the variable list. If therde are not enough data in the buffer to satisfy the entire variable list, the file will be read again as necessary to complete the list.

If INPUT# is reading to a numeric variable from a random-access (file described later in this chapter) and the record being read from is either empty or contains only non-numeric information, a

```
?TYPE MISMATCH ERROR
```

message is displayed.

BASIC allows an INPUT# statement with no variables in the variable list, but such a statement does nothing.

You can use the OPEN statement to access a directory or subdirectory, just as you would to access a text file. Thus, you could use INPUT# statements to access a directory and obtain, one line at a time, the same information displayed by the CATALOG statement. BASIC converts the next directory entry into an 80-character text line each time input is requested from a directory file.

If an INPUT# statement calls for a numeric variable, but the input buffer does not contain numeric data, BASIC will display a

```
?TYPE MISMATCH ERROR
```

message. If the input buffer contains numeric characters followed by nonnumeric characters in that line, the numbers are accepted, the other characters are discarded, and BASIC displays a

```
?EXTRA IGNORED
```

warning message.

The OUTPUT# statement

Normally, BASIC sends all its output, such as error messages and prompts, to the video screen. If you want to **redirect** output to another file, perhaps to get a record of a program's output, you can use an OUTPUT# statement.

OUTPUT# redirects screen output to a specified file. BASIC will send all PRINT, LIST, TRACE (but not TRACE TO #), and CATALOG statement output to the specified file, but error messages and keyboard input are still echoed to the screen. You specify the file used for output by its file reference number (set by an OPEN statement) following the reserved word OUTPUT#. For example:

```
)OUTPUT #5
```

will send output to file #5

If there is no file open with the given file reference number, BASIC displays the message

```
?FILE NOT OPEN ERROR
```

If the file specified is not a file that can accept characters, BASIC displays the message

```
?TYPE MISMATCH ERROR
```

To resume normal screen output, type

```
)OUTPUT# 0
```

and BASIC will again display characters on the screen.

The TRACE statement should not be used with the OUTPUT# statement. Use the TRACE TO # option, discussed in Chapter 1 in the section "Debugging Programs," to direct trace output to a character or disk file.

❖ *Note:* INPUT prompt strings will be sent to the file specified in the OUTPUT# statement, not to the screen.

The PRINT# statement

PRINT# writes text characters to files in the same way that PRINT writes information to the screen. PRINT# is followed by the file reference number, a semicolon, and a list of expressions separated by commas. Optionally, you can specify a record number by following the file reference number with a comma and the record number. In this case, BASIC will start writing information to the file at the beginning of the specified record.

Here are some examples of PRINT# statements

```
)PRINT# 1; W$(0,0,0), LEFT$(W$(0,0,1))
)PRINT# 10, 4755; A&+24, T&/43, R%
```

PRINT# automatically performs any necessary numeric-to-string type conversions and transfers the text characters to the file. Numbers are formatted in either fixed-point or floating-point notation, according to the same rules used by the PRINT statement (that is, SHOWDIGITS controls the format of numbers generated by PRINT#).

You can use the SPC specification with PRINT# statements in the same way that you use them with PRINT statements. (See the "TAB and SPC Specifications" section of Chapter 3 for details.)

Warning:

Although PRINT# allows commas in place of semicolons (as PRINT does), they may cause unexpected breaks to appear in the output because files have no tab positions. You can also use the TAB specification, but it too may cause strange results. Another allowable, but not recommended, practice is to run some expressions together without commas or semicolons.

The PRINT# USING statement, which controls the format of text characters sent to a file, is described in detail in Chapter 3, in the "Formatting Information" section.

Accessing data files

READ# and WRITE# statements are used for accessing BASIC data (BDF or DATA) files. Data file access is much faster than text file access because no text-to-binary conversion is required. The advantage of a text file is that it allows you to use PRINT# and INPUT# statements, which are usually the most convenient way to handle text input and output.

The READ# statement

READ# gets information from a data file, specified by its file reference number. Optionally you can include a record number to specify a particular record for BASIC to start with in a random-access file. A variable list following the file reference number (and optional record number, if included) defines where to put the information being read. For example:

```
)READ# 7; Pip1, Pip2
)READ# 8, 54; Twelve%, Strong&(2)
```

If you specify a record number, the first field in the specified record in the file is assigned to the first variable in the READ# statement.

BASIC stores the information in a BDF file one variable at a time, in binary for numeric variables and as a string for string variables. Each variable begins with a **tag** byte that defines both the type and size of the binary or string information that follows. Each variable (also called a **field**) must fit entirely within a record; a field may not span a record boundary within the file.

When BASIC opens a BDF file, it allocates a record buffer, where all file input or output is actually done first, then it transfers the entire record to or from the disk file media. BASIC only does the actual reading or writing of records when necessary, not for every READ# statement.

READ# automatically performs any type conversions needed for numeric data. However, it does not automatically perform type conversions between numeric data and string variables (and vice versa), and an attempt to read a string with a numeric variable (or vice versa) results in a

```
?TYPE MISMATCH ERROR
```

message.

The following table defines the conversion limits of the READ# statement.

Variable to data field type	**Result**
Real to:	
Real	OK
Double real	OK, with possible loss of accuracy
Integer	OK
Double integer	OK, with possible loss of accuracy
Long integer	OK, with possible loss of accuracy
String	TYPE MISMATCH ERROR
Double real to:	
Real	OK
Double real	OK
Integer	OK
Double integer	OK;
Long integer	OK, with possible loss of accuracy
String	TYPE MISMATCH ERROR
Integer to:	
Real	OK in the range of ±32K, else OVERFLOW
Double real	OK in the range of ±32K, else OVERFLOW
Integer	OK
Double integer	OK in the range of ±32K, else OVERFLOW
Long integer	OK in the range of ±32K, else OVERFLOW
String	TYPE MISMATCH ERROR

Double integer to:

Real	OK in the range of ±2E+9, else OVERFLOW
Double real	OK in the range of ±2E+9, else OVERFLOW
Integer	OK
Double integer	OK
Long integer	OK in the range of ±2E+9, else OVERFLOW
String	TYPE MISMATCH ERROR

Long integer to:

Real	OVERFLOW ERROR if more than ±9E+18
Double real	OVERFLOW ERROR if more than ±9E+18
Integer	OK
Double integer	OK
Long integer	OK
String	TYPE MISMATCH ERROR

String to:

Real	TYPE MISMATCH ERROR
Double real	TYPE MISMATCH ERROR
Integer	TYPE MISMATCH ERROR
Double integer	TYPE MISMATCH ERROR
Long integer	TYPE MISMATCH ERROR
String	OK

The message

```
?FILE TYPE ERROR
```

is displayed if you attempt to use the READ# statement with a file that is not a BDF (or DATA) file. (The file type descriptors BDF and DATA are synonyms for the same ProDOS file type attribute.)

The WRITE# statement

WRITE# sequentially writes the binary value of each variable or constant in its expression list to a field in a specified data file. Follow the word WRITE# with the file reference number, a semicolon, and a list of expressions separated by commas. Optionally, you can specify a record number for BASIC to begin with by following the file number with a comma and the record number. If you specify a record number, the value of the first expression in the expression list is written to the first field in the specified record. Otherwise, records are written sequentially.

Here are some examples of WRITE# statements:

```
)WRITE# 3; Major%, Minor%, Xlow
)WRITE# 4, 11; Map(1,3,5,7,9)
```

Each field in a data file consists of a tag byte that defines the type and size of the information in that field, followed by the value information. Each field is just large enough to contain the tag byte and the binary or string data, and so the fields are of variable length. Any specific type of numeric field always has a fixed length, and a string field has a variable length.

WRITE# does not perform numeric-to-string conversions while transferring information from the expressions to the file, it just writes a binary image of numeric data to the file.

If a file record lacks enough room for all the fields being written to it, the extra fields will be written to the next record. Note that writing any new data to a record will cause the old data in that record to be lost.

If you try to write a data field to a file that is longer than the file's record length, BASIC displays the message

```
?OUT OF DATA ERROR
```

Important:

Note that the usual rules for determining the type of an expression result are in effect. An expression may have an integer, a double-integer, a long-integer, or an extended-precision real result. Whatever type an expression returns, that type will be written to the file, except an extended-precision result will be converted to a double-precision real, and then written.

If you want to ensure that a given field in a record is a specific type, you must use a variable instead of an expression, or force the expression result to the required type with one of the CONV functions. For example:

```
)WRITE #1;N%+1
```

writes an integer.

```
)WRITE #1;N%+1.23
```

writes a double-precision real.

Sequential and random access

There are two ways to access text and data files on a disk: **sequential access** and **random access**. Sequential access is like reading a book; accesses begin at the front of the file and continue on toward the end. Random access requires that the file be made of equal-sized records, and it means that you can access any record in any order. Character devices may not be accessed randomly, but block type devices, such as disks, allow either form of access.

Sequential access

Here is an example of a sequential access program:

```
10 REM Program PrintSequential
20 FILE$ = "SequentialText"
30 OPEN FILE$, FILTYP= TXT FOR OUTPUT AS #10
40 FOR X=1 TO 10
50 PRINT #10; "This is line ";X
60 NEXT X
70 CLOSE #10
80 END
```

This PrintSequential program writes both string and numeric values into a sequential text file.

Line 20 assigns the string SequentialText to the string variable FILE$. Line 30 opens (and creates, if necessary) the file and assigns the number 10 to it as a file reference number. As long as the file is open, it is referred to as #10. Lines 40 and 60 define a loop that will execute 10 times. Each time through the loop, X has a different value: first 1, then 2, and so on up to 10.

Line 50 writes two values into file #10 each time it executes. The first value is the string "This is line", and the second is the character string representation of the numeric value of X. These two strings are joined together (because of the semicolon between them) to occupy one line of text in the file.

Line 70 closes the file. In a larger program, other routines might need to access files, and unless closely controlled, problems arise with attempts to operate with more than six or seven files open at one time or by accessing the wrong file.

After the program runs, the contents of the file are:

```
This is line 1
This is line 2
This is line 3
•
•
•
This is line 10
```

To see the contents of this file on the screen, you need another program:

```
10 REM Program InputSequential
20 FILE$ = "SequentialText"
30 OPEN FILE$, AS #11
40 ON EOF #11 GOTO 80
50 INPUT #11; ACCEPT$
60 PRINT ACCEPT$
70 GOTO 50
80 CLOSE #11
90 END
```

The InputSequential program opens the file to read its contents. Each time the loop in lines 40 through 70 executes, line 50 reads the next line of text from file #1, and stores it in the string variable ACCEPT$. Then line 60 displays the string on the screen.

The SequentialText file in the previous example was a text file because of the way it was opened. You can use a BASIC data file to achieve the same result. The following program creates a data file and writes some data into it:

```
10 REM Program WriteSequential
20 FILE$ = "SequentialData"
30 CREATE FILE$, FILTYP=DATA
40 OPEN FILE$, FOR OUTPUT AS #12
50 FOR X=1 TO 10
60 WRITE #12; "This is line ",X
70 NEXT X
80 CLOSE #12
90 END
```

The WriteSequential program is like the PrintSequential program, but it uses WRITE# instead of PRINT#. WRITE# does not allow the use of semicolons to separate values that are written to a file. Each WRITE# sends a field to the file for each variable in its expression list. WRITE# also does not convert numbers to strings, but places them in a file using the same format (binary) as numeric variables that are stored in the computer's memory.

In the SequentialData file, every other field contains the string "This is Line", and the fields in between contain binary coded numeric values from 1 to 10.

The program below reads information from the SequentialData file back into memory and displays it on the screen:

```
10 REM Program ReadSequential
20 FILE$ = "SequentialData"
30 OPEN FILE$, FOR INPUT AS #13
40 FOR X=1 TO 10
50 READ #13; ACCEPT$,INNUM
60 PRINT ACCEPT$; INNUM
70 NEXT X
80 CLOSE #13
90 END
```

Where the InputSequential program used INPUT#, the ReadSequential program uses READ#. The READ# statement reads the values of two fields from the file: the first (a string value) is stored in ACCEPT$, the second (numeric) value is stored in INNUM. In line 60, PRINT displays the two values as one line because of the semicolon.

ReadSequential displays the following on the screen:

```
This is line 1
This is line 2
.
.
.
This is line 10
```

Sequential access considerations

Notice that the programs above with PRINT#, INPUT#, WRITE#, and READ# statements all use sequential access; we never had to specify where to begin in the file. PRINT# or WRITE# statements cause one access for each expression in their expression list, and INPUT# or READ# statements cause one access for each variable in their variable list. BASIC automatically advances by one data or text item each time an expression is written or a variable is read, so that the next access to the file will be positioned correctly.

When you open a file, the first access begins at the beginning of the file. Each subsequent access begins where the last one left off.

This means that each time that you open an existing file and write to it, at least some of its original contents will be written over. If all of the original file is not written over, where is the end of the file? At the end of its original contents, or at the end of its new contents? The answer is that you can't be sure. If the old contents have been fully overwritten, they are lost. If not, a portion of the old contents will remain after the end of the new contents. BASIC will not tell you either way.

To avoid problems with the old contents of files, don't open an existing file using a FOR OUTPUT statement for sequential access. Instead, delete the old file, then open a new file using the same file reference number with the FOR OUTPUT statement. (Before you delete the old file, be sure that you read any information you need from it.)

If you just want to append information to the end of a file, open it using the FOR APPEND statement, When an existing file has been opened with FOR APPEND, the first access begins at the end of the file. Each subsequent access begins where the last one left off. This allows you to retain information previously saved in the file.

Random access

Random-access files are structured as a sequence of equal-sized records. In random-access operation, you specify exactly where each file access should begin by specifying a record number in the PRINT#, INPUT#, WRITE#, and READ# statements.

The CREATE and OPEN statements allow you to specify the record size of a new file. If you don't specify it, the record size defaults to 512 bytes. You cannot change the record size of an existing file.

Note that in sequential access, the record size is irrelevant; you do not need to think about it. A record in a sequential file cannot be accessed randomly.

In a data file, the contents of a record are organized into fields. Each field contains either a numeric or string value. Any given field is always contained wholly within one record, the records do not overlap.

In a text file, each record is a series of bytes; with each byte containing a character.

To use random access, you must have a clear idea of how information will be organized in your file. In a data file, you should usually plan to have each record contain the same kind of fields. For example, each data record might contain two real values, an integer, and a string, in that order. The record size must be large enough to contain all the fields you will write in each record.

Each data type uses a certain amount of bytes in its field, as follows

Data type	**Bytes used**
Real	5
Double real	9
Integer	3
Double integer	5
Long integer	9
String	string length + 2

If you want to randomly access text in a text file, remember that the INPUT# statement reads lines of text ended with a return character. Therefore, you should usually plan to have each record contain one line. This means that the record size should be at least big enough to contain the longest line your program will ever write into it. A line requires 1 byte for each character in the line, plus 1 byte for the return character at the end of the line.

We can modify the sequential access examples so that they use random access to display only the even-numbered lines of the file on the screen.

To change the PrintSequential program to the PrintRandom program, we must specify a record size in the CREATE statement and a record number in the PRINT# statement.

```
10 REM Program PrintRandom
20 CREATE "RandomText", FILTYP=TXT, 16
30 OPEN "RandomText", FOR OUTPUT AS #10
40 FOR X=1 TO 10
50 PRINT #10,X; "This is line " ;X
60 NEXT X
70 CLOSE #10
80 END
```

Although we could have specified a larger record size, 16 bytes is enough to contain the longest string we will be writing— *This is line 10* is 15 characters plus 1 for the return character at the end.

In line 50, X is the record number. Each time through the loop, the PRINT# statement writes to a different record; first it writes to record 1, then record 2, and so on, up to record 10.

We can change the InputSequential program to the InputRandom program, without specifying the record size, because BASIC stored that information when the file was created. When you open the file BASIC retrieves the record size.

However, we do need to make two changes: one in the FOR statement and another in the record number in the INPUT# statement.

```
10 REM Program InputRandom
20 OPEN "RandomText", FOR INPUT AS #11
30 FOR X%=2 TO 10 STEP 2
40 INPUT #11,X%; ACCEPT$
50 PRINT ACCEPT$
60 NEXT
70 CLOSE #11
80 END
```

The FOR statement now starts with X%=2 and has STEP 2, causing X% to take on the values 2, 4, 6, 8, 10 as the loop repeats five times. In the INPUT# statement, we specify X% as the record number; so the first time through the loop we access record 2, the second time record 4, and so on, up to record 10.

When you run InputRandom, it displays the following on the screen:

```
This is line 2
This is line 4
This is line 6
This is line 8
This is line 10
```

In the same fashion, program WriteSequential can be changed to a program to randomly access data files. Try it!

Random access considerations

Here are the essential rules for using random-access files:

- When a record number is specified in a file I/O statement, the access begins at the beginning field of that record.
- When you overwrite any part of an existing record using a WRITE# (not PRINT# statement), all the previous content of that record is lost. However, an existing record that is not overwritten remains unchanged.
- When accessing data files, if a READ# statement contains more than one variable or a WRITE# statement contains more than one expression, the current record position will move from one field to the next.
- When accessing text files, if a PRINT# statement writes more than one line, each is placed in the file in the order written, regardless of record boundaries. If an INPUT# statement reads more than one line, it assumes that each one begins where the last one left off, regardless of record boundaries.
- If a field in a data file won't fit in the space remaining in the record, BASIC goes to the beginning of the next record. If a field is too big to fit in any record, BASIC displays the message

```
?OUT OF DATA ERROR
```

File statements and functions

BASIC contains statements that allow you to control program execution according to information contained in files that your program accesses.

The ON EOF# statement

You can use ON EOF# to force BASIC to allow your program to control what happens if BASIC reads past the end of a file (EOF stands for end-of-file). When ON EOF# is not in effect, and BASIC reads past the end of a file, it displays the message

```
?OUT OF DATA ERROR
```

and halts execution. ON EOF# is very similar to the ON ERR statement, except that ON EOF# recognizes only the end-of-file event.

Follow the reserved word EOF# with a statement or statement list, and execution will branch to that statement list whenever BASIC reads past the end of the file instead of displaying an error message or halting execution. For example:

```
) 100 ON EOF #10 PRINT "End of file"
) 1000 ON EOF #12 GOTO 2000
```

The statement list is executed as though a GOTO statement had caused execution to jump there. Unlike with ON ERR statements, RESUME does not function in conjunction with ON EOF# statements.

The OFF EOF# statement

OFF EOF# cancels an ON EOF# statement. After an OFF EOF# statement has been executed, BASIC resumes displaying error messages and halting execution when it reaches the end of a file, just as it did before the ON EOF# statement was executed. You must follow the reserved word EOF# with a file reference number to specify which file's ON EOF# statement should be canceled.

The reserved variable EOF

When BASIC encounters the end of a file, it assigns the file reference number of the file causing the error to the reserved variable EOF. You can then check the reserved variable EOF to determine which file ran out of data.

When you use the reserved variable EOF in an ON ... GOTO or ON ... GOSUB statement, you must enclose EOF in parentheses. For example:

```
)ON (EOF) GOTO 100,200,300
```

The EOFMARK function

EOFMARK returns the current value of the end-of-file mark for the file specified by the value of it argument, which can be any arithmetic expression in the range of 1 through 29. This function is only valid for open block device (disk) files. If you reference a file that is not currently open, BASIC displays the message

```
?FILE NOT OPEN ERROR
```

If you use the EOFMARK function with a character device file, you will see the message

```
?NOT A BLOCK DEVICE
```

The FILE function

FILE tests the existence of a disk filename given by its first parameter, a string expression. It returns the value 1 if the file with the given pathname exists, or the value 0 if the file does not exist. The FILE function also allows an optional second parameter, separated from the string expression by a comma.

The second parameter is the reserved word FILTYP= followed by a file type descriptor or a numeric expression whose value is in the range 0 through 255. The valid file type descriptors are TXT or TEXT, SRC, BDF or DATA, and DIR or CAT.

If any error other than

```
?FILE NOT FOUND ERROR
```

is encountered, that error will be displayed. If you do not specify a file type, BASIC returns a true (the value 1) for a file of any type. If you specify a type different from the one the file already has, BASIC will display

```
?FILE TYPE ERROR
```

The reserved variable AUXID@ will contain the subtype from the directory entry of the file.

The FILTYP function

FILTYP returns the file type of an open file from the BASIC FCB. Its argument is the reference number of the file. FILTYP(0) is a special case that returns the file type of the last FILE function call.

FILTYP has the same error conditions as the EOFMARK and TYP functions (described below).

The TYP function

You can use TYP to determine what type of data will be read from a particular file on the next access to that file. TYP only works for BASIC data files (FILTYP=BDF). If the file is not a BASIC data file, BASIC displays the message

```
?FILE TYPE ERROR
```

The argument to the function can be any arithmetic expression, but its value must specify a particular file reference number from 1 through 29. If you use a larger or smaller argument, BASIC displays the message

```
?ILLEGAL QUANTITY ERROR
```

The number returned by the TYP function denotes what type of data will next be read from the specified file. TYP actually returns the value of the tag byte of the next field in the file. For example:

```
) 2000 ON TYP(3) GOSUB 2010,2100,2200,2300,2400,2500,2600
```

means that the next item is a double integer.

For a BASIC data file, TYP returns the following values:

0 End of file
1 Not used
2 Integer
3 Double integer
4 Long integer
5 Single real
6 Double real
7 String

If there are no more data items in the file, TYP returns the value 0.

If you specify the reference number of a file that is not open, BASIC displays the message

```
?FILE NOT OPEN ERROR
```

The REC function

REC returns the current record number of the file specified by the value of it argument, which can be any arithmetic expression whose value is in the range of 1 through 29. REC returns a number in the range of 0 through 8,338,607.

If you use the INPUT# or READ# statements to access a window file, REC returns the Window Manager window-record pointer. If you use the INPUT# statement to access a directory file, REC MOD 65536 gives the line number in the last catalog line generated.

REC has the same error conditions as the TYP function, described above.

Example of file I/O

Here is an example of file use. Assume that you have inserted a BASIC diskette named APPLE1 in disk drive 1 and turned on the power. Assume that there is no initialization file (GSB.HELLO program), so no program is run automatically; you just see the prompt character. Now you type

```
RUN DEMO
```

to load and run the DEMO program, show below. This program makes copies of existing files.

```
10 PRINT "Text file copy utility"
20 INPUT "Enter input file pathname: ";A$
30 OPEN A$, FILTYP=TXT FOR INPUT AS #10
40 INPUT "Enter output file pathname: ";A$
50 REM Open new output file
60 OPEN A$, FILTYP=TXT FOR OUTPUT AS #11
70 ON EOF #10 PRINT "Done" : CLOSE : END
80 INPUT #10; A$ : PRINT #11; A$ : GOTO 80
```

Line 10 displays a message on the screen. Line 20 displays a prompt and then waits for you to enter a pathname. You must enter a legal pathname, or you will see a message, and the program will halt. Line 30 opens the named file, referenced hereafter as #1. Line 40 asks for another pathname; and line 60 opens the file for output, assigning to it the file reference #2. Line 80 performs the actual copying. Line 70 will be executed only when the end of the input file has been reached to end the program. Until line 70 is executed, line 80 reads in lines from the input file, and writes them to the output file.

Recall that when you boot your computer, the volume name of the diskette in the built-in disk drive is stored in prefix 0. This means that a valid pathname can be as little as the name of the file (assuming that there are no subdirectories on the diskette). So, responding simply **AFILE** and **BFILE** to the program prompts would cause /APPLE1/AFILE to be duplicated to a new file named /APPLE1/BFILE. Remember that not using a slash before a pathname causes the contents of prefix 0 to be added to the beginning of the pathname.

This program can be used to print a file by responding **AFILE** and **.PRINTER,** assuming that a printer is properly connected. Responding **.CONSOLE** and **/APPLE1/TEXT** will take input directly from the computer's keyboard and write it to the new file /APPLE1/TEXT created by the program.

Note that in the last case, where input is taken from the keyboard, the program will not terminate normally because block devices (.CONSOLE and .PRINTER) have no end of file. (In the next-to-last case, .PRINTER was used for output, not as an input file.) You can use Control-C to end the program.

Chapter 6

External Routines

The toolbox interface xx

Toolbox definition files xx
The LIBRARY statement xx
The CALL statement xx
The EXFN_ statement xx

The invokable module interface xx

The INVOKE statement xx
The PERFORM statement xx
The EXFN statement xx

Apple IIGS BASIC provides two complete interfaces to external subroutines (assembly-language procedures or functions that are not part of IIGS BASIC). Each interface allows you to load and execute external subroutines from BASIC programs. You can use both external subroutines that you write and those included with your Apple IIGS.

For example, there is a tool set of subroutines for displaying graphics, supplied in the Apple IIGS read-only menory (ROM). This tool set, called QuickDraw II, can display graphics objects of many types in the Super Hi-Res video display mode.

If you catalog your IIGS BASIC master diskette, you will see some files with the extension .INV. These are one type of external routines. There are two other groups of related files that you should be aware of, both having the same name as the external routine files. One type has the extension .DOC (for documentation). The other type of files, which have no extension, are BASIC demonstration programs. If you run a file with a .DOC or no extension, it will describe the use of its associated external routines.

A subroutine is a separate part of a program called by one or (usually) several other parts of the program to perform a specialized or frequently repeated task. A subroutine may have a list of arguments enclosed in parentheses following its name, either variables, pointers, or expressions. There are two types of external subroutines: procedures and functions. A function returns a value, a procedure does not.

An external subroutine uses its parameter list to tell the calling program what information is available for its use, what is to be operated on, or where it should leave the results of its operation for the use of the calling program.

The Toolbox interface

BASIC supports an interface to the Apple IIGS Toolbox tool sets, some of which are provided on the System disk. You can access these tool sets by using the BASIC statements LIBRARY, CALL, and EXFN_.

Tool sets are of two types: read-only memory (ROM) and random access memory (RAM). ROM tool sets are built into the Apple IIGS computer, and RAM tool sets are loaded from the System disk. RAM tool sets are always kept in a special sub-subdirectory on the boot or System disk. You can view these files by typing

```
CAT  */SYSTEM/TOOLS
```

The files in this directory all have names that follow the pattern TOOL*999*, where the *999* is the tool set number from 001 to 255.

❖ *Note:* You can write your own tool set. See the *Apple IIGS Toolbox Reference* manual for details.

Toolbox definition files

The LIBRARY statement loads a special file, called a **TDF** or **toolbox definition file,** that contains a dictionary of interface definitions for all the functions and procedures in a tool set. Each interface definition contains the function or ligrary name name, tool number, function number, parameter count, and parameter type for each procedure or function in the tool set.

A complete set of TDF files is supplied with BASIC for all the standard Apple IIGS Toolbox ROM and RAM tool sets. The individual procedures and functions for all the tool sets are documented in the *Apple Toolbox Reference* manual. Appendix H provides more information about TDF file format.

As far possible, the names of the procedures and functions in the TDF dictionaries are the ones used in the *Toolbox Reference* manual; certain tool sets have duplicate names, so some function names in the TDF will not match those in the *Toolbox Reference* manual.

The LIBRARY statement

The LIBRARY statement loads one or more TDF files into the BASIC library dictionary (a separate memory segment allocated for interface definitions). The reserved word LIBRARY is followed by one or more string expressions separated by commas. Each string expression must be the pathname of a TDF file on a currently mounted disk volume.

If BASIC does not have enough free memory for all the dictionary data, it displays the message

```
OUT OF MEMORY ERROR
```

If no filenames are present, the library dictionary is deleted, except for entries inserted by the INVOKE statement.

When you use the LIBRARY statement with just a pathname parameter, BASIC discards all prior library definitions before loading the new ones. If you want to add a TDF file to the existing library dictionary without deleting the currently loaded entries, use the APPEND option. For example:

```
)LIBRARY APPEND "TDF.QUICKDRAW"
```

will append the dictionary for the QuickDraw II tool set to the library segment.

The header record inside the TDF file may request that the tool set be loaded from disk. In this case, use the Tool Locator LOAD1TOOL call to load the tool set from the TOOLS sub-subdirectory of the System disk. IIGS BASIC first checks to verify that the same TDF file definitions have not already been loaded into the library dictionary; if the TDF file was loaded previously BASIC skips that file and processes the next TDF filename (if any).

Thus, you can safely reexecute a LIBRARY statement any number of times while testing a program from immediate mode, and only load the TDF definitions once. Note, however, that the TDF file must continue to be accessible since BASIC will open the file and read the header record read before making the duplicate load check.

The CALL statement

CALL executes a named procedure in an Apple IIGS tool set. Before you can use CALL for normal functions, most tool sets must be properly initialized. All tool sets have a startup function that must be called before using any other functions.

Calling a procedure is done like this:

```
1100 CALL CLEARSCREEN(-1)
```

or

```
1200 _ClearScreen(BK%)
```

The second example shows the use of the shorthand CALL verb, the underscore character (_). The CALL and _ statements in both examples call the ClearScreen function in the QuickDraw II tool set, which clears the entire Super-Hi-Res screen, using the value of the parameter to set a word, or 2 bytes, of pixels.

Each procedure or function in a tool set has a function number and a tool number, along with its parameter requirements. All three of these items (and others) are extracted from the interface definition entry in the library dictionary. The dictionary entry is found by searching for the libname, in this case, CLEARSCREEN.

The TDF file for the tool set must have been loaded into the library dictionary with the LIBRARY statement prior to executing a CALL libname; otherwise, BASIC will display the message

```
UNDEF'D PROC/FUNCTION ERROR
```

The dictionary entry indicates the parameters required by the tool set function (and their order and types). The parameter list in the CALL statement must contain the proper number, order, and types of arguments within parentheses following the libname. The parameters are pushed on the CPU stack in order from left to right, and the proper tool set function is called.

BASIC removes any returned results from the CPU stack and stores the first 16 words (32 bytes) in the return stack buffer. The contents of the return stack buffer may be accessed through the R.STACK functions.

Warning:

Don't attempt to use CALL without complete knowlege of a tool set. (The standard tool sets are documented in the *Apple IIGS Toolbox Reference manual.)*

IIGS BASIC correctly initiates QuickDraw II when the GRAF INIT command is executed, and it also starts up the Sound Manager (but not the NOTESYN or NOTESEQ tool sets). You may obtain the addresses of some preallocated memory resources that are useful for initializing certain tool sets with the BASIC@ function.

To pass real or integer numbers or the values of variables, just include them in the argument list as an expression (for an explanation of expressions see the sections titled Expressions and Statements in Chapter 2). If the type of the numeric argument or expression you use does not match the type of the parameter required by the tool set function, CALL attempts to convert the result to the proper type, just as if you had used the proper CONV( ) function for the type of the argument.

Warning:

BASIC will not perform string-to-numeric or numeric-to-string conversions; in these cases, an Argument Type Mismatch Error will occur.

You must also use the correct number of arguments when calling a tool set function; otherwise, you will see the message

```
?ARGUMENT COUNT ERROR
```

- *Note:* The binary format of real numbers are those defined by the SANE tool set. If an expression is used for a parameter, the expression evaluation may create an extended-precision real result, which will be converted to the type required by the tool set function. This conversion may cause an Overflow Error if the result of the expression is a number too large for the type of parameter required by the function.

To pass the **address** of a numeric variable, use the VARPTR function. There is no means of passing the address of an expression.

If the tool set interface definition entry obtained from the library indicates that the argument for a tool set function should be a counted string (often referred to as a Pascal string or P-string), the CALL statement will convert a BASIC string, or string expression result, into a counted string. A **counted string** is a count byte followed by the characters. CALL automatically passes the address of the count byte to the function instead of the address of the BASIC string. You do not need to use the VARPTR function for P-string parameters in the tool sets defined by the standard TDF, since all of these definitions were set up in advance to use the counted string conversion function described above.

Warning:

The counted string conversion will only pass strings up to 254 characters long. Attempting to pass a string with 255 characters (the limit case) will cause a String Too Long Error.

To pass the address of the string's first character (without a count byte) use the VARPTR$ function. See Chapter 8, "BASIC Reference," for more details.

The EXFN_ statement

EXFN_ executes tool set functions that return a numeric value. The library dictionary, loaded by the LIBRARY statement, contains the libnames that EXFN_ can call.

The name of the external function must follow the reserved word EXFN_. EXFN can be used anywhere in a BASIC statement that a variable can be used. For example:

```
10000 PRINT EXFN_StringWidth("THIS IS A SAMPLE")
```

You can use one of the following type characters to document the type of the function result, even though the function result type is actually controlled by the interface definition entry information in the library dictionary:

```
% @ & $ #
```

The type character immediately follows EXFN_. For example, the statement

```
10000 PRINT EXFN%_StringWidth("THIS IS A SAMPLE")
```

indicates that the result of the external function is a regular integer.

If BASIC does not find the libname in the library dictionary, it displays the message

```
?UNDEF'D PROC/FUNCTION ERROR
```

If you want to pass an integer argument, just include an integer variable in the parameter list, but as a variable, not as an expression.

To pass the address of numeric variables, use the VARPTR function. String variables are converted to counted strings and the address is passed for the argument. EXFN_ processes arguments in the same manner as the CALL statement.

Additional technical details about EXFN_ can be found in Chapter 8, "BASIC Reference."

The invokable module interface

The BASIC statements, INVOKE, PERFORM, and EXFN provide the interface to user-written external subroutines, called invokable modules. These statements are used to load a file containing external subroutines into memory from disk files and execute them at the BASIC program's demand.

Invokable modules are similar to tool sets, but are user-written and are specifically dependent on the internal operation of the IIGS BASIC interpreter. (A tool set, even a user-written one, is normally coded to be independent of the calling environment.) How to write an invokable module is described in Appendix I.

The INVOKE statement loads external subroutines and their dictionaries with the System Loader tool set and, depending on the subroutine's type, either the PERFORM or EXFN statement executes it.

The INVOKE statement

INVOKE loads into memory the files whose names are given by the string parameters following the reserved word INVOKE. For example, to load an invokable file named FastPrint, enter

```
)INVOKE FastPrint
```

You may load as many files at once as you like by separating the pathnames by commas. The following is an example of INVOKE used in immediate mode:

```
)INVOKE FP1, FP2, /Vol2/Subr/FP3
```

Using INVOKE in deferred mode is somewhat different; the filenames must be string constants or string variables, like this:

```
110 FILNAM3$="/VOL2/SUBR/FP3"
120 INVOKE "FP1","FP2",FILNAM3$
```

Executing INVOKE with just a list of filenames discards from memory any subroutine modules previously loaded by other INVOKE statements and returns the freed memory space to the Memory Manager. You can add to the existing set of invoked modules by using the APPEND option with INVOKE. For example:

```
200 INVOKE APPEND "Banner.Printer"
```

will add the subroutines in the named file to those already loaded. INVOKE APPEND does not discard the previously invoked modules (if there are any) before loading the new module.

Invokable modules are written in assembly language using the Apple IIGS Programmer's Workshop (APW) Assembler, following the guidelines found in Appendix I. An invokable module must be a load file in object module format. The file must have a data segment with the segment name DICTIONARY and a code segment containing the external subroutines.

The dictionary segment of an invokable module contains an interface definition for each procedure and function subroutine used in the code segment. A single code segment may have from 1 through 255 entry points defined within its dictionary. BASIC loads the dictionary through first, and adds the entries to the invoke table within the library dictionary. The code segment is loaded with the System Loader.

❖ *Note:* If you don't need your invoked subroutines any longer and want to free the memory, execute an INVOKE statement with no pathnames following it. All the invoked subroutines are removed from memory, variables defined in BASIC are not touched, nor is the BASIC program altered. The freed memory is returned to the Memory Manager and will not be available for BASIC variables or arrays unless the data segment size is expanded with the CLEAR statement.

If there is not enough memory to load an invoked file, BASIC will display the message

```
?OUT OF MEMORY ERROR
```

If the file loaded is not a load file with file type S16 ($B3), BASIC will display the message

```
?FILE TYPE ERROR
```

If the file is not found on the named disk, you will see the message

```
?FILE NOT FOUND ERROR
```

The PERFORM statement

PERFORM executes a named external procedure previously loaded by an INVOKE statement. If an argument list is present (enclosed in parentheses after the procedure name), each argument is evaluated and passed to the procedure before execution. Numeric arguments are converted to the type specified by the interface definition entry in the INVOKE dictionary.

The library dictionary contains the names of the procedures that may be performed. The dictionary entry also contains a description of the number, order, and type of arguments required by the procedure. The INVOKE statement reads the library dictionary entry from the dictionary segment of the invokable load file when the assembly-language module is loaded.

To pass real or integer constants or the values of single variables, just include them in the argument list. A string or string expression may not be used for a numeric argument or vice versa; attempting to do so will display the message

```
?ARGUMENT TYPE MISMATCH ERROR
```

If the proper number of arguments is not supplied, you will see the message

```
?ARGUMENT COUNT ERROR
```

To pass addresses of variables, use the VARPTR function. For example:

```
)PERFORM Errproc(R,13-6,VARPTR(D))
```

passes the value of variable R, the value 7, and the address of the variable D to the procedure named Errproc.

If you want your subroutine to operate on a BASIC string in memory, simply using a string variable will pass an address pointing to the string's descriptor in memory. The subroutine should be designed to act on the string using the address of the descriptor. Alternately, you may pass the address of the string data by using the VARPTR$ function.

A third choice is also available if you define the argument as a counted string argument in the interface definition. When a counted string argument is required by the procedure, IIGS BASIC creates a counted string from a BASIC string and passes the address of the count byte as the argument.

Values passed to an external subroutine are pushed on the system stack in memory. When the routine is executed, it must read the values from the stack.

Addresses of variables to be passed to an external subroutine are pushed on the system stack by BASIC only if the VARPTR function is used. It is the responsibility of the subroutine to distinguish between variable values and addresses. Only single variables can be used a the argument of the VARPTR function; using an expression is not legal.

Let's say we have two subroutines that each take one argument. The first one, MyProc, takes the value of a real expression. The other one, MyOtherProc, takes the address of a real variable. The following are examples of various legal and illegal combinations of arguments to these subroutines:

```
)PERFORM MyProc(4.5) : REM Legal: A simple expression
)PERFORM MyProc(NUMS) : REM Value of NUMS is passed
)PERFORM MyProc(NUMS+4.5) : REM NUMS+4.5 is legal a expression.
)PERFORM MyOtherProc(VARPTR(NUMS)) : REM passes address of NUMS
)PERFORM MyOtherProc(VARPTR(4.5)) : REM Illegal use of VARPTR
```

The EXFN statement

EXFN executes an external assembly-language function that returns a value and has been loaded by an INVOKE statement. (EXFN_, with an underscore, is used to call tool set functions.)

As with EXFN_, you can use one of the following type characters to document the type of the function result, even though the actual type of the result is controlled by the interface definition entry information in the invoke dictionary.

```
% @ & $ #
```

For example, suppose that you have a function named CalcX that performs some operation on a supplied argument and returns a double-precision result of the operation. You could execute CalcX in immediate mode with the following statement:

```
)PRINT  EXFN#CalcX(2)*32/256
```

The value returned by CalcX is multiplied by the expression 32/256. Remember that the argument passed to CalcX is contained within the parentheses following the function's name.

The rules for passing arguments to external procedures also apply to external functions. See the previous sections on the EXFN_ and PERFORM statements for details.

If the function named is not part of a currently invoked file, you will see the message

```
?UNDEF'D  PROC/FUNCTION  ERROR
```

Chapter 7

Advanced Topics

Procedures xx

Using procedures xx
Defining procedures xx
Argument passing xx
Local and global variables xx

Functions xx

Using functions xx
Defining functions xx

Memory management xx

Memory segments xx
- The user data segment xx
- The program segment xx
- The library segment xx
- Record buffer segments xx
- Invoke segments xx
- Tool set segments xx

Using the CLEAR statement xx
Using the NEW statement xx
Using the FREEMEM function xx
Memory management errors xx

The INPUT USING statement xx

IMAGE statement parameters xx
- The *maxlen* parameter xx
- The *cursorx* and *cursory* parameter xx
- The *scrnwidth* parameter xx

The *fillchar* parameter xx
The *cursor-mode* parameter xx
The *long* and *short* parameter xx
The *modmask* parameter xx
The *control* parameter xx
The *immediate* parameter xx
The *beep* parameter xx
The *bord-char* parameter xx
The *spare* parameter xx
The *n-chars* parameter xx
The *tchar* parameter xx
The *tmodfr* parameter xx
The *tmode* parameter xx
The UIR function xx

Using task master xx

Prerequisites xx
Setting up the enviorment xx
Using the EVEDEF statement xx
Event and menu handling routines xx

Opening a window file xx

Using ON EXCEPTION statements xx

Apple IIGS BASIC supports several advanced programming capabilities that require detailed explanation. In addition, some statements require information only documented in other Apple IIGS Technical Library publications.

The following topics are discussed in this chapter:

- procedures
- multiline functions
- memory management
- the INPUT USING statement
- the Task Master call and Window and Menu Managers
- window files
- ON EXCEPTION event-trapping

The beginning programmer need not master any of these features, but reading the sections on procedures and functions can be helpful after learning the fundamentals.

Procedures

Procedures are groups of BASIC statements similar to subroutines, with the added advantages of speed of execution and modularity.

This can have multiple formal arguments and local variables that are isolated from the rest of the program. This separation allows a subprogram to define its own local variables with the same names as variables in the main program and retain separate values. Procedures can also be used to create side-effect multiline string functions by returning a global string variable as a result.

The use of local variables allows modular program design. You can use procedures in multiple programs without considering variable duplication. Generic procedures can make up a library of tools that can be integrated into new programs.

BASIC scans a program once, during the RUN and CHAIN commands, for all the procedure definitions and inserts the names into the variable table. When a procedure is executed, BASIC searches the variable table instead of the entire program for its definition; thus, a procedure will usually start execution faster than a subroutine invoked with a GOSUB statement.

Using procedures

You use the PROC statement to reference procedures, which can have an argument list with one or more arguments.

Arguments or real parameters are the values of the variables used in the PROC statement that are passed to a procedure during execution. For example:

```
PROC  DrawIsoTriangle(XLEFT,YLEFT,XRIGHT,YRIGHT,66)
```

uses the arguments XLEFT, YLEFT, XRIGHT, YRIGHT and the constant 66. The values of the arguments are passed to the procedure, and their values are assigned to the parallel formal parameters in the procedure.

The term formal parameter refers to the parameter(s) that are defined in the DEF PROC statement. For example, the DEF PROC statement for this procedure might look like this:

```
DEF  PROC  DrawIsoTriangle(X1,Y1,X2,Y2,HEIGHT)
```

The formal parameters are the local variables X1, Y1, X2, Y2, and HEIGHT. They become local variables within the procedure, and the values of the arguments become the initial values of the local variables. The formal arguments do not return their values to the arguments; the passing is one way into the procedure. This parameter-transfer approach is called **pass by value.** This means that a procedure cannot change the value of the argument by changing the value of the formal parameter.

In IIGS BASIC, a procedure can reference both its local variables and all the global variables of the main program, as long as there is not a local variable with the same name as the global variable. Local arrays are not supported, and all array references are global.

Defining procedures

You begin a procedure definition with a DEF PROC statement and complete it with an END PROC statement. The DEF PROC statement must be the first statement on a program line. A procedure returns control to the next statement after the calling PROC statement when an END PROC statement is executed. A procedure may have more than one END PROC statement, but there must be at least one END PROC statement at the beginning of a line following the DEF PROC statement.

The general syntax for defining a procedure is a follows:

```
1000 DEF PROC procedure-name [ (formal-parameter-list ) ]
1010 LOCAL variable list
   •
   •
   •
1090 END PROC [procedure-name]
```

The *procedure-name* must follow the syntax of a variable name, and it can be up to 29 characters long. This name cannot be duplicated in any other DEF statement as the name of a procedure or function, and it cannot have a type character on the end.

The *formal-parameter-list* is optional, and it can only contain simple variables. Parameters are separated by commas, and they can be any type of simple variable. The number of parameters is limited by the length of a program line (239 characters), but a large number of parameters will execute slowly, since they are created each time a procedure is executed.

Parameters do not retain their values between invocations of a procedure. If your procedure needs variables that retain their values, you must use global variables or arrays.

Each formal parameter in the list becomes a local variable when a procedure is invoked and has the value of the matching argument stored in it as its initial value. You can define additional local variables with the LOCAL statement. These will have initial values of zero or null.

The statements between the DEF PROC and END PROC statements are called the **body** of the procedure.

You cannot use a user-defined function definition (DEF FN ... END FN) or another procedure definition in the body of a procedure.

Even though procedure and function definitions cannot be nested, a procedure can *reference* another procedure or function within its body. When a procedure calls another procedure, the inner procedure cannot reference the local variables of its caller, unless they were passed as arguments, and it cannot change the local variables of its caller.

Argument passing

Arguments used in PROC statements can be any type of simple variable, array element, or constant. A string variable or expression must be passed for a string formal parameter, and a numeric variable or expression must be passed for a numeric formal parameter. If you attempt to pass a string argument to a numeric formal parameter or vice versa, BASIC will display the message

```
?TYPE MISMATCH ERROR
```

BASIC will convert the type of a numeric argument to the type of the numeric formal parameter if they do not match. This conversion may cause an overflow error if the value of the argument is out of range for the type of the numeric formal parameter.

Local and global variables

When you assign a definition to a variable within a procedure without using a LOCAL statement in the procedure definition BASIC will create a global variable, just as it does when you make an assignment in the main program. You can avoid this by using the FN variable = assignment statement for variables within a procedure or function. This FN LET statement will only assign definitions definitions to local variables, and will display a

```
?NOT LOCAL ERROR
```

if you unintentionally use a global variable name.

A procedure can share a variable with the main program by simply not declaring it in a LOCAL statement. Arrays are always shared and global to all procedures, functions, and the main program.

Functions

You can define two types of user functions in IIGS BASIC: single-expression (or simple) functions and multiline functions. Both can have multiple formal arguments, and multiline functions can have additional local variables, which are isolated from the rest of the program..

BASIC scans a program once, during the RUN and CHAIN commands, for all the function definitions and inserts the function names into the variable table. (It does not scan for the definition when you use a GOTO statement to execute a program.) When a function is referenced, BASIC searches the variable table for the function entry and executes the function through the resulting program location pointer.

Using functions

You reference single-expression, or simple, functions in the same manner that you reference variables. You can use a simple function anywhere within your program that a simple variable can be used. Functions must have an argument list with one or more arguments. See the "Using Procedures" section earlier in this chapter for definitions of the terms *formal parameters* and *formal arguments*.

Multiline functions are referenced like variables, but only in the expression of a LET or FN = assignment statement.

Both multiline and simple functions are referenced by preceding the function name with the reserved word FN. For example,

```
20 A = FN RECIP(X)
```

could be a reference to a simple or multiline function, but the statement

```
30 PRINT FN RECIP(xyz)
```

could only be a reference to a simple function.

Defining functions

An introductory explanation of simple functions is given in Chapter 2, "Tools of Your Trade." Simple functions are defined with a single program statement, which must be the first statement on a program line. Functions may not be defined in immediate mode, but you can refer to a function before defining it within your program. Single-expression functions can be of any type numeric type. As with variable types, function types are selected by including a type character.

You can also define single-expression string functions (but not multiline string functions) using the following syntax:

```
20 DEF FN name [%'@'&'#] (var{,var}) = arithmetic expression
30 DEF FN name $ (var{,var}) = string expression
```

You must include a parameter list, enclosed in parentheses, with at least one parameter. Numeric and string functions can have either numeric or string parameters of the same or different type as the function result. However, a numeric function definition must be have a numeric expression, and a string function definition must have a string expression.

A function can have as many parameters as will fit in one program line, but the more parameters a function has, the slower it will execute.

A multiline function definition is begun with a DEF FN statement and finished with an END FN statement. The DEF FN statement must be the first statement on a program line. A multiline function returns a numeric result to the referencing LET expression when the END FN statement is executed. Multiline string functions are not allowed.

A multiline function can have more than one END FN statement, but you must include at least one END FN statement as the first statement on a line following the DEF FN line.

The general syntax for defining a multiline function is as follows:

```
DEF FN function-name [%'@'&'#](formal-parameter list )
[ LOCAL variable-list ]
   •
   •
   •
FN function-name = expression
END FN function-name
```

The *function-name* must follow the syntax of a variable name, and it can be up to 29 characters long. This name cannot be duplicated in any other DEF statement as the name of a procedure or function.

The *formal-parameter-list* is required, and it can only contain simple variables. Parameters are separated by commas, and they can be any type of simple variable. The number of parameters is limited by the length of a program line (239 characters), but a large number of parameters will execute slowly, since they are created each time a function is executed.

Parameters do not retain their values between invocations of a function. If your function needs variables that retain their values, you must use global variables or arrays.

Each formal in the parameter list becomes a local variable when a function is invoked and has the value of the matching argument stored in it as its initial value. You can define additional local variables with the LOCAL statement. These will have initial values of zero or null.

The statements between the DEF FN and END FN statement are called the **body** of the function. You cannot use a user defined procedure definition (DEF PROC ... END PROC) or another function definition (definition nesting is prohibited) in the body of a function. Furthermore, certain statements, such as a DIM statement, may not work more than once if they occur within the body of a function (the second reference to the function may cause a duplicate definition error). However a DIM statement could be executed *conditionally* in a function. For example, you could dimension an array within an IF statement as follows:

```
IF MYTABLE%(0)=0 THEN DIM MYTABLE%(55) : MYTABLE%(0)=1
```

This technique works because a reference to an undefined array does not cause automatic dimensioning of the array (only a LET assignment to an undefined array element causes automatic dimensioning). Any reference to an undefined array element returns a zero (or a null string for string arrays).

Warning:

This technique will probably not work with a compiler if a IIGS BASIC compiler becomes available in the future.

Even though procedure and function definitions may not be nested, a multiline function may reference another procedure or function within its body. When a function calls another procedure or function, the inner function cannot reference the local variables of its caller, unless they were passed as arguments, and it cannot change the local variables of its caller.

The discussions of argument passing and local and global variables in the "Procedures" section of this chapter also apply to simple and multiline functions.

Memory management

With Apple IIGS BASIC, you can easily create large programs for the Apple IIGS and dynamically control the allocation of the user data segment and the program segment during program execution. IIGS BASIC uses the Memory Manager tool set to allocate memory segments, as described in this section.

Memory segments

The Memory Manager allocates three main memory segments. The CLEAR and NEW statements provide memory management control for the user data segment and the program segment, respectively. It allocates a third segment for the library dictionary segment, which can be deallocated by the CLEAR INVOKE and CLEAR LIBRARY options of the CLEAR statement.

IIGS BASIC does not automatically try to allocate all available memory when you start up the interpreter. Generally, the interpreter code data segment requires all of one 64K bank of memory, and allocates a small initial userdata segment of 32K before attempting to run the GSB.HELLO program. All the BASIC interpreter is loaded into memory as a single code data segment upon initial startup.

The three main memory segments are all allocated in multiples of 256 bytes so changing the size of one of these segments may not produce the exact results you requested. File buffers are always allocated in word (2-byte) multiples, and thus will contain 1 extra byte for odd record lengths.

- *Technical note:* The intepreter keeps all its Memory Manager segments locked, and has numerous dereferenced pointers to the partitions within them, until one of the three main segments must be resized. Then three main segments (and sometimes all the file buffer segments) are unlocked and one of the main segments is resized. The Memory Manager may then move one or more of the unlocked segments during memory compaction. Following resizing, all the pointers are re-dereferenced after locking all the segments. Only the record buffer handles are dereferenced on-the-fly during BASIC programs.

he user data segment

The user data segment can be as large as necessary, but it has a functional limit derived from the sum of the size of all arrays, simple variables, and 64K for string data plus the free memory required for the transient local variable tables for functions and procedures. Generally, the maximum user data segment size would be the sum of memory for arrays plus simple variables plus 128K.

The user data segment contains speed partitions. In order of the lowest to highest address, these are

- the array partition
- the simple-variable partition
- the local-variable-stack partition
- free memory
- the string-data pool
- the invoke-control partition

The free-memory partition provides the only gap between the user data segment partitions. The string-data pool and invoke-control tables grow downward into the gap, and the lower partitions grow upward into the gap. The FRE reserved variable returns the size of the gap (the free memory partition within the user data segment.)

Even if the FRE variable returns a small number, there may be large amounts of unallocated memory available outside the interpreter memory segments. To use this memory in a BASIC program, you must first expand the user data segment with the CLEAR statement (see the discussion of the CLEAR statement in Chapter 8).

Dimensioning or erasing an array with a DIM or ERASE statement moves the variable and local partitions; erasing a variable and reference or assignment to a new simple variable moves the local-variable partition. Executing the INVOKE statement will move the string pool downward.

In addition to the above actions, opening a file can shrink and/or move the user data segment when a file record buffer must be allocated or when ProDOS requires more unallocated memory to open a file. Also, many functions within the Apple IIGS Toolbox require that unallocated memory be available for their operation.

If you are planning to write a BASIC program with large memory requirements, remember that you probably cannot expand the user data segment to consume all the memory in the Apple IIGS. If you are going to be writing a program using the Window, Menu, Control, and/or Dialog managers, they will require 128K to 192K, plus work space, depending on how many managers you use and what you do with them.

In addition to the three main segments, a separate Memory Manager segment is allocated for the record buffer of each open file (see further under "Record Buffer Segments" later in this section). Also, the System Loader allocates each invoked module as a separate memory segment.

Warning:

The fact that the memory segments may move during program execution presents a problem for programmers familiar with Applesoft and other BASICs, in which the addresses of the user data elements remain constant during program execution. Even though the VARPTR function is provided in IIGS BASIC, the addresses returned by it are not fixed and can become stale if they are not immediately used and then discarded.

Using the VARPTR function on a local variable is very likely to become stale because allocating a global simple variable will move all the stacked local variables.

IIGS BASIC will shrink the user data segment when it needs to expand the program or library segments, allocate a new record buffer, free space for ProDOS to allocate an I/O buffer, or add interface definitions to the library segment for an invokable module during an INVOKE statement. Thus, it is safe to over-allocate the size of the user data segment within the limits discussed earlier in this section.

The program segment

The program segment contains the program header partition, the program-text partition, and a free-memory partition (for adding new program lines).

When you load or run a program, BASIC expands the program segment, using unallocated memory first, to accommodate the program. When you load a program in immediate mode, BASIC expands the program segment to 112.5 percent of the size of the program's EOFMARK (shown in the CATALOG display as the ENDFILE field). The extra 12.5 percent of space is provided for editing the program.

The addition of lines beyond this initial 12.5 percent margin causes the program segment to be expanded as required, in portions of at least 256 bytes at a time. Loading or expanding a program may cause the user data segment to move and/or shrink if enough unallocated memory is not available from the Memory Manager.

BASIC will not reduce the program segment size when it needs memory for other purposes or if a smaller program is loaded, run, or chained. Once the program segment expands to a given size, it will remain that size unless you use the NEW statement with the size option. Using a size of 512 bytes with the NEW statement will shrink the program segment to a size just larger than the program.

IIGS BASIC will not shrink the program segment when it needs to expand the library segment, allocate a new record buffer, free space for ProDOS to allocate an I/O buffer, or add interface definitions to the library segment for an invokable module during an INVOKE statement. Thus, it is not safe to over-allocate the size of the program segment.

The library segment

The Memory Manager allocates 512 bytes to the library segment during interpreter startup, and it remains this size if the LIBRARY and INVOKE statements are never used. When you load a TDF file with the LIBRARY statement or invoke a module with the INVOKE statement, BASIC expands the library segment as required.

The library segment contains three partitions: the TDF partition, the invoke partition, and a free partition. The TDF partition, which is used by the LIBRARY statement, contains a linked list of TDF tables, one per LIBRARY file. The invoke partition contains a single interface definition table (IDT) with the same format as a TDF table. The IDT is expanded dynamically by the dictionary segment of each invoked module loaded by the INVOKE statement.

IIGS BASIC will not shrink the library segment when it needs to expand the program segment, allocate a new record buffer, or free space for ProDOS to allocate an I/O buffer. The size of the library segment cannot be explicitly allocated from BASIC, but it expands as needed during INVOKE and LIBRARY statements. You can reduce the library segment by using the CLEAR INVOKE or the CLEAR LIBRARY (see Chapter 8 for details).

Record buffer segments

Each open file, except for file types TXT and SRC, has a record buffer allocated as a separate Memory Manager memory segment. The handle for the memory segment is stored in the IIGS BASIC file control block (FCB). Whenever the record buffer is referenced (read or written), the handle is dereferenced into a work pointer and the data transferred between the buffer and the variable tables.

If you do not specify a record size in the OPEN statement, BASIC assigns a size of 512 bytes and allocates the buffer accordingly. For other record sizes, an even number of bytes is always allocated, even if the record size is odd. Thus, the memory segment may be 1 byte longer than the record's size. This is done to minimize the time required to fill the buffer with zeros during buffer allocation.

Invoke segments

Each time a module is invoked, BASIC uses the System Loader to load the code segment into memory, thus creating a memory segment. The System Loader information about each of these segments is retained in the invoke-control partition at the top of the user data segment. For each segment loaded by an INVOKE statement, a 10-byte record is created and stored in the table so the segment can be referenced and deallocated as required.

You can deallocate invoke segments by using the INVOKE or CLEAR INVOKE statements. BASIC automatically deallocates all invoked segments when a QUIT statement is executed.

Tool set segments

You can use the LIBRARY statement to load a RAM tool set, but BASIC does not deallocate the segments that are allocated by the LOAD1TOOL call or when a QUIT statement is executed. You must deallocate memory segments and shut down any tool sets that you load during a BASIC program if the memory they use is to be returned to the Memory Manager. Refer to the *Apple Toolbox Reference* manual for further details about using tool sets in your programs.

Using the CLEAR statement

You can use the CLEAR statement to expand and shrink the size of the user data segment during the execution of a program. If you combine the use of ERASE and CLEAR statements, your program can dynamically add and then remove arrays or change the user data segment size in overlays invoked through a CHAIN statement.

The CLEAR statement is discussed in detail in Chapter 8. The syntax of the CLEAR statement is

```
CLEAR user-data-size
```

BASIC treats the *user-data-size* argument as a request to expand or shrink the user data segment to the number of bytes given. During interpreter startup, this size defaults to approximately 32K. You can create a GSB.HELLO program that allocates more or less if you want to select another size for normal use.

Unless you use the CLEAR statement, the user data segment will not expand or contract, even when you clear or expand the data within the segment. For example, attempting to dimension a large array without first expanding the user data segment can result in an out of memory error.

The *user-data-size* argument cannot be less than 8192 or larger than the amount of memory currently available through the Memory Manager. Because the user data segment is a contiguous memory block, memory fragmentation may make it impossible for the Memory Manager to allocate all available memory into the user data segment. A detailed discussion of memory fragmentation can be found in the *Apple Toolbox Reference* manual, chapter on the Memory Manager.

Your programs can adjust memory usage for many purposes, using differing amounts at various times, with the CLEAR statement. Note that the CLEAR statement with the size option does not delete or change the arrays or variables that are allocated, but will cause the string-pool partition to be compacted. The ERASE statement, described in Chapter 8, removes an array or variable from the appropriate partition within the user data segment and enlarges the free-memory partition, but does not shrink the user data segment.

You can use the FREMEM function, (discussed later in this chapter) to find out how much memory is allocated or free for various partitions within the memory segments, as well as how much remains unallocated and available to the Memory Manager.

Using the NEW statement

You can use the NEW statement to expand and shrink the size of the program segment during the execution of a program. If you combine the the use of CHAIN and NEW statements, the program segment can dynamically grow and shrink with overlays invoked through the CHAIN statement.

The NEW statement is discussed in detail in Chapter 8. The relevant syntax of the NEW statement for memory management purposes is:

NEW *program-size*

BASIC treats the *program-size* argument as a request to expand or shrink the program segment to the number of bytes given. During interpreter startup, this size defaults to 512 bytes.

If the *program-size* argument is smaller than the current program, BASIC will use the smallest size that will contain the program instead of the size requested. Using the NEW statement is the only way to shrink the program segment once it has grown due to loading or chaining to a large program.

NEW first attempts to expand the program segment without stealing space from the user data segment. If unallocated space is available from the Memory Manager, the user data segment remains unchanged. If space is not available, IIGS BASIC will attempt to shrink the user data segment to deallocate enough memory to expand the program segment to the requested size. If it cannot use space from the user data segment, NEW will display the message

```
?OUT OF MEMORY ERROR
```

and leave the program segment as it was (although it may have moved).

Using the FREMEM function

The FREMEM function provides useful information about the partitions of the user data segment and the program segment, as well as the other memory segments allocated by the Memory Manager. The syntax of the FREMEM function is:

FREMEM(*n*)

The FREMEM(*n*) syntax provides 10 different function results, as follows:

`FREMEM(0)` Returns the free memory in the user data segment, without first performing the compaction to recover unused string space

`FREMEM(1)` Returns the size of the user data segment after performing compaction to recover unused string space. The number will usually be 256 larger than the reserved variable FRE when all arrays and variables are clear

`FREMEM(2)` Returns the amount of memory currently allocated for arrays within the user data segment

`FREMEM(3)` Returns the amount of memory currently allocated for simple variables (not including local variables) within the user data segment

`FREMEM(4)` Returns the size the current program will have when it is saved on disk, including the size of the program header

`FREMEM(5)` Returns the size of the program segment

`FREMEM(6)` Returns the size of the library segment

`FREMEM(7)` Returns the Memory Manager's unallocated memory total (and does a CompactMem without unlocking any BASIC memory segments)

`FREMEM(8)` Returns the size of the Memory Manager's largest free contiguous block.

`FREMEM(9)` Returns the total memory installed in the system, excluding the 64K Digital Oscillator Chip (DOC) RAM dedicated to the sound generator

By subtracting the FREMEM(0) result from the FREMEM(1) result, you can monitor the "high water" mark of memory usage during program execution. At frequently used places within your program, you could capture the FREMEM(0) value into a variable whenever it is smaller than the prior value of that variable, thus recording the smallest free memory available in your program. This will let you know whether you have over-allocated the user data segment size and should reduce it to free memory for other purposes. Generally, you should set the user data segment size from 10 to 20 percent above the high-water mark determined using this method.

❖ *Note:* Remember that the string-pool partition will never grow to more than 64K in size, even if FREMEM(1) indicates more space is available.

By summing the results of FREMEM(2) and FREMEM(3), you can determine the static variable memory requirement for a program, after all variable and arrays have been allocated and dimensioned. This sum, plus space for transient local variables and 64K for the string-pool partition, is the maximum useful user data segment size for a given program. This total will only be a constant if you are not using the ERASE statement.

There are three 64K banks (192K) in the Apple IIGS that are essentially reserved for use and control by the ProDOS operating system, the System Loader, the Memory Manager, and the video display buffers.

The IIGS BASIC interpreter requires most of another 64K bank, thus consuming 256K of the minimum 384K required to run IIGS BASIC. This leaves about 128K for programs, data, record buffers, and so forth in a minimum Apple IIGS system. Add to this the 128K to 192K required if all the RAM-based Graphics, Window, and Desktop tool sets (with their data) are used, and you will need a minimum of 512K.

A large BASIC application program using the full toolbox could easily use a 768K system. Add to this a future Finder with multiple resident copies of IIGS BASIC, and you will be getting out of memory errors in a 1 megabyte system.

Memory-management errors

Here are some of the most common causes of out-of-memory errors

- dimensioning a large numeric array without first expanding the user data segment with the CLEAR statement
- loading and attempting to use all the toolbox tool sets in a 512K system with a large program
- allocating a very large (more than 64K) program segment with a NEW *program-size* statement (it is not necessary to preallocate the program segment since it will automatically expand in small increments as needed)
- attempting to run IIGS BASIC under a Switcher after loading numerous other large applications (IIGS BASIC requires at least 160K of free memory with a contiguous 64K segment for the interpreter and a 32K segment for the user data segment)

IIGS BASIC may also display the message

```
?TOOLSET ERROR =$02xx
```

when an unexpected Memory Manager error occurs or a toolbox function is unable to allocate needed memory. These errors are defined in the *Apple IIGS Toolbox Reference* manual chapter about the tool set you called, and in Chapter X, "Memory Manager."

The INPUT USING statement

INPUT USING executes a user input routine (UIR) using the parameters in an IMAGE statement. The UIR is the same routine used by the EDIT command and for entering command lines in IIGS BASIC. You can customize the behavior of the input routine for your programs with the IMAGE statement parameters. The parameter *linnum1* or *label1* points to the IMAGE statement.

The IMAGE statement for INPUT USING is similar to the one used for PRINT USING, but instead of specs for each variable, it contains a fixed format sequence of initial parameters. The string variable, *svar,* is both input to and output from the INPUT USING statement. The value of the string is the default value of the line to be edited by the user; it may be a null string if new data are being entered. The edited characters of the line (if any) are returned in place of the default value.

The UIR function provides the status information from the UIR after the INPUT USING statement completes. The UIR function is described later in this chapter.

IMAGE scanning parameters

The parameters in the IMAGE statement are separated by commas, and all the parameters up through *tmode1* are required. The parameter *n-chars* is a count from 1 to 10 that indicates how many sets of termination character definitions follow.

The order and names of the IMAGE statement parameters for INPUT USING are as follows:

IMAGE *maxlen, cursorx, cursory, scrnwidth, fillchar, cursor mode, short, long, modmask, control, immediate, beep, bord-char, spare ,n-chars,*$tchar_1$*,* $tmodfr_1$*,* $tmode_1$[... ,$tchar_6$, $tmodfr_6$, $tmode_6$... $tchar_{10}$,$tmodfr_{10}$,$tmode_{10}$]

The *maxlen* parameter

The **maxlen** parameter indicates the maximum length of the result string. Enter a number from 1 through 255. If the maximum size of the input string is larger than *scrnwidth,* the width of the field on the screen, the UIR uses the invisible part of the input string to save characters that were pushed out of the field by insertions. Thus, *maxlen* may be greater than *scrnwidth.* However, in this case, the length of the result string actually returned by INPUT USING statement is limited to **scrnwidth** characters.

The cursorx, and *cursory*

The *cursorx* and *cursory* parameters contain the relative coordinates of the start of the field within the current textport. When the UIR is entered initially (not reentered after an interrupt termination), *cursorx* and *cursory* are used to position the cursor at the beginning of the input field on the screen display. Select values for x from 1 through 80; select values for y from 1 through 24.

The *scrnwidth* parameter

The *scrnwidth* parameter tells the UIR how wide to make the field on the screen. When the UIR is called, it displays the input string's default value at the cursor position defined by *cursorx* and *cursory*. If there is any room left in the field, fill characters are displayed (the number of fill characters equals the *scrnwidth* minus the length of the input string). You can set *scrnwidth* from 1 through 254.

If the value of *scrnwidth* is greater than the number of character positions from the start of the field (as defined by **cursorx** and **cursory**) to the end of the textport minus two, the UIR reduces *scrnwidth* to the maximum available less two.

The *fillchar* parameter

The *fillchar* parameter determines the character that is used to fill the unused portion of the field. Normally, *fillchar* is set to space, enter 32, or " ". If *fillchar* is any value less than 32, the MouseText underline (MouseText I) character is used as the fill character.

The *cursor-mode* parameter

The *cursor-mode* parameter indicates which cursor mode is being used. Set it at 0 for the insert cursor, and at 1 for the replace cursor. Control-E toggles between the two cursor types. The initial value is normally 0 (insert mode), but your application program can force the UIR to start with the replace cursor by setting this parameter to 1.

The *long* and *short* parameters

The *long* and *short* parameters are the countdown values used to create the correct blinking frequency for the cursor. The nominal values for long are in the range of 200 through 800 and about half these number for short.

An important part of the *Human Interface Guidelines* is that the cursor blinks 80 times a minute, with one phase taking twice as long as the other. That is, if the insert cursor is active and under a character in the input field, the character should be visible twice as long as the underline. If the replace cursor is active, the inverse character should be visible twice as long as the normal character.

The values that you select for *long* and *short* control the cursor blink rate. However, if you activate immediate mode, the cursor will no longer blink at the correct rate because your invokable assembler program will get control in the middle of the blink loop. In this case, your IMAGE parameters for *long* and *short* must be adjusted so that the cursor will again blink at the desired rate.

The *modmask*

The *modmask* (for modifier mask) parameter is used to cause the UIR to ignore meaningless or unwanted bits in the keypress modifier byte derived from the modifier word returned by the Event Manager. The UIR uses 8 bits extracted from the Event Manager modifier word as the UIR modifier. The *modmask* parameter is ANDed with the UIR modifier byte before comparing it with the termination character modifier list (*Tmodfr*).

The bits of the UIR modifier byte and *modmask* are defined as follows:

Table 7-1

UIR bit	On add value	Bit description	Event Manager modifier word
7	128	KeyPad bit	Bit 13
6	64	Control key	Bit 12
5	32	Option key (closed Apple)	Bit 11
4	16	Caps Lock key	Bit 10
3	8	Shift key	Bit 9
2	4	Apple key (open Apple)	Bit 8
1	2	Btn0State NOT	Bit 7
0	1	Btn1State NOT	Bit 6

❖ *Note:* The Btn0/1 state bits are inverted from their state in the Event Manager modifier word.

For general use, the following bits should be 0, or off: Btn0State, Btn1State, Shift key, Caps Lock key, and Control key. The KeyPad bit should also normally be 0 unless you are using the keypad keys as termination characters (as function keys of some type). Most termination characters are usually produced by Apple key or Apple and Option key combinations.

Various conflicts arise if you do not mask out a given bit. For example, the Control key bit is not set when the Return key is pressed, but it is set if the user types Control-M. Either method will return the ASCII character code 13, but with different modifiers.

Unless you want Control-M to be treated differently than the Return key, the Control key bit in *modmask* must be 0. In the same manner, the Enter key on the keypad returns the same ASCII code as the Return key, and if you want Enter to function like Return, the KeyPad bit must be 0. When the KeyPad bit is enabled, you may need to define separate termination character entries for the Return and Enter keys.

The normal value to use for *modmask* is 36 (32+4). When using KeyPad termination characters, a value of 164 (128+32+4) should be used.

The *control* parameter

If the *control* parameter is initially set at 0, control characters (ASCII values less than 32) are not allowed as input (typing a control character causes a beep).

If you set this parameter to 1, control characters are allowed as input from the keyboard. To insert a control character, the user must press the Option key, the Control key, and one other key. This lets the user type, for example, Option-Control-X as an input chararcter and still use Control-X as an editing command.

The actual value inserted in the string, during editing, is the ASCII value of the letter key plus 128, which appears on the screen as the inverse of the corresponding character. For example, to insert the carriage return character (ASCII 13), the user presses Option-Control-M. The screen shows an inverse M, and the result string will contain the real value of Control-M,13.

The result string is scanned for characters greater than 128, and BASIC converts them into the proper control codes before returning the string to your program.

Note that editing and termination characters are not affected by the setting of control.

The *immediate* parameter

The *immediate* parameter will normally be 0. Selecting immediate mode by setting this parameter to 1 enables the external vectoring through the INPUT USING statement immediate mode vector. The address of this vector is obtained through the BASIC@ function. When immediate mode is enabled, the UIR routine calls an external assembler routine after every keypress.

To use immediate mode, you must write an invokable module yourself and link the invokable module to the UIR by storing the address of your invokable module entry point into the vector. Refer to Appendix I for more information about how to write invokable modules.

If you accidently enable INPUT USING immediate mode without setting up the vector first, the UIR will act as if you have not enabled immediate mode because the default address in the vector returns directly to the UIR routine.

Specialized filtering and sequence checking can be implemented using immediate mode, for example, date entry that checks the date against the month and leap day against the year. The external routine is called with a JSL instruction and must return through an RTL instruction.

The *beep* parameter

Set the *beep* parameter to 0, and any illegal keypresses will cause the UIR to beep. If it is set to 1, there is no beep.

The *bord-char* parameter

Normally, the cursor blinks by alternating between the cursor character and the space character or a data character in the field. When the field is filled and the cursor resides one character beyond it, *bord-char* (border character) is used instead of space. To use a blank space as the border character, enter 32 or " ".

The *spare* parameter

The *spare* parameter is not in use at this time. You must enter 0.

The *n-chars* parameter

The *n-chars* parameter specifies the number of termination characters you want to define. When any character is typed during INPUT USING data entry, the character is checked against the list of termination characters you supply with this parameter and the ones that follow. Typing a termination character will signal completion or temporary interruption of data entry and return control from the UIR to your BASIC program. For example, you would set it at 2 if the only termination characters you want to use are Return and Escape. If you are using other termination characters, you must set *n-chars* accordingly.

The *n-chars* parameter must be a number from 1 through 10, and it must be followed by exactly that number of groups of three parameters each. The three parameters in each group define exactly what keypress is the termination keypress or keypress combination. The first element of each three parameter group is the ASCII code of the termination character, described as *tchar* below. The second parameter is a bit mask that defines which of eight possible modifier bits must occur with the ASCII code, as described under *tmodfr* below, and the third parameter is the exit mode, either terminate or interrupt, for that termination character definition.

The *tchar* parameter

Termination characters are the ASCII codes that terminate input and cause the UIR to return with the resulting string of characters typed by the user. Examples of termination characters include the Return key, the Esc key, and Apple key-? (for help). The *tchar* parameter can be entered as a number from 1 through 127, as any single character (other than a digit), or as a character in quotation marks. For example, M, 77, and "M" are all valid and equivalent *tchar* parameters.

The *tmodfr* parameters

Each *tmodfr,* (for termination modifier) parameter is paired with the corresponding *tchar* ASCII code to define exactly which keypress or keypress combination is a termination character that will terminate UIR input and return control to your BASIC program.

The values to use for *tmodfr* are defined in the table shown below. When a given *tmodfr* bit is enabled, that bit must be on in the UIR modifier, after ANDing with the *modmask* parameter, for the current keypress to match and thus terminate input. If you want multiple bits, sum the enable values in the table for the appropriate bits, and enter the total as the *tmodfr*.

Table 7-2

tmodfr bit	Enable value	Bit description	Required state
7	128	KeyPad bit	Keypad keypress
6	64	Control key	The Control key must be pressed
5	32	Option key	The Option key must be pressed
4	16	Caps Lock key	The Caps Lock key must be down
3	8	Shift key	The Shift key must be pressed
2	4	Apple key	The Apple key must be pressed
1	2	Btn0State NOT	Paddle button 0 must be pressed
0	1	Btn1State NOT	Paddle button 1 must be pressed

❖ *Note:* the Btn0/1 state bits are inverted from their state in the Event Manager modifier word.

The *tmode* parameter

The *tmode* parameter is normally 0 and thus defines the termination character as a terminate character. When *tmode* is set to 1, the termination character is treated as an interrupt character that will temporarily suspend editing of the input field and return control to your program. Interrupt mode is designed to allow you to implement external editing features in your program.

For example, you could define Apple key-? as an interrupt character and display a help screen describing the UIR editing features, then restart the UIR editing by executing INPUT USING 0 ; *svar*.

The INPUT USING 0 option (0 is not a valid line number) reenters the UIR with the UIR parameters in their prior state. When the UIR exits because of an interrupt *tmode* character, the reentry flag is set so INPUT USING 0 will correctly restart the editing process. The cursor is returned to its prior position and mode, as are the other elements of the editing process.

The UIR function

The UIR function returns the status information from the UIR after an INPUT USING statement completes.

The UIR function returns the following status results:

UIR(0) Returns the exit_type that terminated input editing. Exit_type is the index of the *tchar* (termination keypress) that caused the termination. It will be in the range of 1 to *n-chars*. When immediate mode is enabled, exit_type will be 0 when the external assembler routine is called by the UIR; this indicates that the UIR has not yet terminated.

UIR(1) Returns the ASCII value of the termination or immediate mode keypress. This will normally be a value from 32 through 126 in immediate mode or the termination character ASCII value, such as 13 for Return or 27 for Esc. This value is used on return to the UIR when INPUT USING 0 restarts an interrupt mode edit.

UIR(2) Returns the UIR modifier (before masking with *modmask*) for the termination or immediate mode keypress. This value is used on return to the UIR when INPUT USING 0 restarts an interrupt mode edit.

UIR(3) Returns the reentry mode status of the UIR parameters. It will be a 1 for an interrupt termination, a 2 after an immediate mode termination.

UIR(4) Returns the last column position of the cursor.

UIR(5) Returns the last row position of the cursor.

UIR(6) Returns field relative position of the cursor upon termination. This function returns a value from 1 through *scrnwidth* and indicates where there cursor was last positioned relative to the data returned in the string variable.

Using Task Master

The Apple IIGS BASIC EVENTDEF, MENUDEF, and TASKPOLL statements and the TASKREC function provide a direct interpreter interface to the Window Manager's Task Master function. This interface allows maximum flexibility for programmers who want to implement window and menu based applications.

Before considering the statements that implement Task Master in BASIC, you should study the following summary of the programming tasks that must be accomplished before TASKPOLL INIT can be executed. This summary describes the general scope and complexity of window-and menu-based applications; it is not a definitive or exhaustive explanation of the tasks.

Prerequisites

If you are considering developing a window-based application program in BASIC with a menu bar, in the classic desktop style of user interface, you should first read the *Human Interface Guidelines* manual. It thoroughly describes the design foundation for this type of application.

You should be familiar with all the introductory material in the Window Manager and Menu Manager chapters of the *Apple IIGS Toolbox Reference* manual and the discussion of Using Task Master, plus the Event Manager chapter. You should also review the definitions of EVENTDEF, MENUDEF, TASKPOLL, and TASKREC in Chapter 8.

Setting up the environment

Before Task Master event polling can be activated, the following tool sets and their associated TDF files must be loaded, in the order shown, through the LIBRARY statement, and then activated in that same order. Then numerous functions need to be called to initialize the environments for each tool set and for Task Master.

1. QuickDraw II (activate by using the GRAF INIT statement in BASIC.
2. Desk Manager
3. Window Manager
4. Control Manager
5. Menu Manager

Additional tool sets that your application may also require include:

6. Font Manager
7. LineEdit Manager
8. Dialog Manager
9. Scrap Manager
10. List Manager
11. Printer Managers (high and low level
12. Standard File Operation
13. Note Synthesizer
14. Note Sequencer
15. FDB

IIGS BASIC uses and has initialized these tool sets during interpreter startup:

- ☐ Tool Locator
- ☐ Memory Manager
- ☐ Miscellaneous
- ☐ Event Manager
- ☐ Sound Manager
- ☐ SANE
- ☐ Integer Math

Each of these managers has from 30 to more than 175 function calls, some of which require complex data tables that you must create. Initialization of these tool sets requires calling from one to a dozen or more of its functions. In addition, if you are going to create an application using a window with scrolling of the content region, the content-scrolling procedure probably must be written in assembly language; attempting to scroll a window by using BASIC statements will usually prove to be unbearably slow.

After starting up QuickDraw II with the GRAF INIT statement in BASIC, you may want to position the cursor, set the background and foreground colors, open a GrafPort as a file, display a message, and show the cursor (mouse pointer).

Desktop environment initialization includes the following steps:

1. Calling _WindStartup to initialize the Window Manager.
2. Calling _Refresh to clear the desktop port.
3. Calling _CtrlStartup to initialize the Control Manager.
4. Calling _LEStartup to initialize the LineEdit Manager.
5. Calling _MenuStartup to initialize the Menu Manager.
6. Calling _NewMenu and _InsertMenu for each menu bar menu.
7. Calling _FixAppleMenu (the Desk Manager) to install desk accessories.
8. Calling _FixMenuBar
9. Calling _DrawMenuBar to display the menu bar on the screen.

Of these tasks, calling _NewMenu is the most complex because you must build a number of interlinked tables to define the content of the pop-up menus. Each menu-definition record contains the text of the menu item, various options, and the menu-item ID number used by BASIC for menu item dispatching. The menu-item numbers you define in the _NewMenu data structures must correlate with the MENUDEF statements in your BASIC program.

After the above tasks, you must define actual application windows for the Window Manager via the _NewWindow function. The _NewWindow parameter list contains 27 parameters, including the addresses of assembler routines and a bit vector of options that defines the type of window frame.

Once the menu bar and window are defined and you have initialized any other tool sets and their data structures, you have completed the initialization of the tool set environment.

Your application may start up without displaying a document window, and thus will need to activate and display a window when the user opens a document file. The Standard File Operation Tool Set provides the standardized dialog box metaphor for locating and opening a document file. Your application would call Standard File Operation functions for the open menu item event from Task Master.

Before you can execute TASKPOLL INIT, you must first execute EVENTDEF and MENUDEF statements enough times, with the proper data, to link the event-handling routines in your program to Task Master through the dispatch tables inside BASIC.

Using the EVENTDEF statement

The EVENTDEF statement is used to store line numbers into the event dispatch table used by IIGS BASIC to direct program control to event-handling routines in your program. This table has 64 entries, numbered from 0 through 63. It can be thought of as a preallocated array into which you must store information. The first 32 entries (0 through 31) have a fixed relationship to the event codes returned by Task Master.

Task Master calls the Event Manager's _GetNextEvent function and passes most of the returned events onto the appropriate functions in the Window Manager and Menu Manager. Task Master can process all but three of the 12 predefined Event Manager events.

Your program will receive the activate keypress and auto-key events for a window. Generally, it will not need to handle any other events.

The following table shows the meaning of the event codes used by Task Master and the event dispatch table:

Table 7-3

Event code	Event Descriptions
0	Null, or no event (BASIC doesn't use this entry)
1	Mouse Button Down event
2	Mouse Button Up event
3	Keypress event
4	---
5	Auto-Keypress event
6	Window Update event
7	---
8	Window Activate event
9	Switch event in the future
10	Desk Accessory event. (not returned to user)
11	Device Driver event
12	application-defined event
13	application-defined event
14	application-defined event
15	application-defined event

The additional event codes shown below are generated when Task Master receives a Mouse Button Down event and calls the Window Manager _FindWindow function to locate where the mouse cursor was pointing when the button was pushed.

Table 7-4

Event code	Event	Location of cursor
16	wInDesk	in the desktop area (not in a window)
17	wInMenuBar	in the system menu bar
18	wInSysWindow	in a system window
19	wInContent	in a window's content region
20	wInDrag	in window-drag (title bar) region
21	wInGrow	in window-grow (size box) region
22	wInGoAway	in window go-away (close box) region
23	wInZoom	in window-zoom (zoom box) region
24	wInInfo	in window-information bar
25	---	in vertical-scroll control
26	---	in horizontal-scroll control
27	wInFrame	in window, but none of the above areas
28	wInSpecial	in special menu item (edit menu)
29	wInDeskItem	? ?

Task Master can handle most of these _FindWindow events; your program will normally need to handle only the wInContent, wInGoAway, and wInInfo events (wInInfo will only occur if your window has an information-bar subdivision). wInMenuBar events are handled through the MENUDEF statement, described later in this chapter.

Generally, you must execute an EVENTDEF statement for the few events that Task Master can't handle and write a routine in your BASIC program to process the event. The EVENTDEF statement is used to inform BASIC of the beginning line number of the event-handling routine. The syntax of the EVENTDEF statement is:

```
EVENTDEF index{,linenum'label}
```

EVENTDEF defines the beginning line number(s) of the event handling routines that will be called after TASKPOLL ON is executed. TASKPOLL on activates the polling for Task Master events. GS BASIC has an internal table of 64 events, 29 of which may be returned by Task Master. The internal table is indexed using the event code number returned by Task Master. If the entry in the table is zero, the event is is discarded (except for event 17).

The *index* parameter is a number from 0 to 63 that defines which event you want to handle in your program, and the is the event handler routine. The index parameter is shown in the left column of the two earlier tables.

The routines referenced by EVENTDEF must end with a RETURN 0 (zero) statement. This special form of the RETURN statement can only be used as the last statement in an event handling routine for Task Master events. RETURN 0 reenables Task Master polling that was suspended when the event-handler routine was called.

EVENTDEF can define sequential events by following the index with multiple line numbers separated by commas. If you want to skip an event, use a *linnum* of zero. Normally, using Task Master only requires defining a few of the 29 possible events; see "Using Taskmaster" in the Toolbox Reference manual.

Event 17 is a special case related to the wInMenuBar events from Task Master. As long as the entry in the EVENTDEF table for event 17 is left a zero, IIGS BASIC assumes that you have defined the menu-item-handling routines with the MENUDEF statement and directs the individual menu-item-select events to their event-handler routines.

Using the MENUDEF statement

The MENUDEF statement is used to store line numbers into the menu-item dispatch table used by IIGS BASIC to direct program control to a menu-item-handling routines in your program. This table has 128 entries, numbered from 0 through 127. It can be thought of as a preallocated array into which you must store information.

When Task Master polling receives the wInMenuBar event from the _FindWindow function, and the resulting menu item is identified by the Menu Manager _MenuSelect function, IIGS BASIC examines entry 17 in the event dispatch table. If the entry is zero, the menu-item dispatch table is indexed with the selected item identification number minus 256. The menu-item dispatch table entry should contain the beginning line number of the menu-item-handling routine.

If entry 17 in the event dispatch table is not zero, the automatic dispatching by the menu-item dispatch table is suppressed, and the wInMenuBar event is dispatched to your program through the line number in entry 17 of the event dispatch table.

When you define your menus and their items with the Menu Manager _NewMenu function, you define the menu item identification number for each selectable item. You must select the identification numbers from the range of 256 through 383 if you are going to use the MENUDEF dispatch mechanism. You need not select them in any order, nor do they need to be sequential. The menu item identification numbers 256 through 383 correspond to MENUDEF index values 0 through 127.

The Menu Manager chapter in the *Apple IIGS Toolbox Reference* manual describes how to create the _NewMenu item and menu definition lines and how to select menu-item ID number in the section entitled Menu Lines and Item Lines.

The syntax and function of the MENUDEF statement is described in detail in Chapter 8.

Event- and menu-handling routines

You can write event- and menu-handling routines in BASIC for many types of applications. Some functions in an application however will be too slow if they are written in BASIC and will need to be written in assembly language. A primary example of this is the scrolling functions needed by a document window with scroll bars. The discussion of the OPEN Window statement later in this chapter presents a general model of what is required to handle the update events for a document window. The Content Definition procedure for a window is required if Task Master is handling update events for your program. However the address of the wContDefProc in the _NewWindow parameter list can't be the line number of a BASIC statement.

IIGS BASIC contains 32 entry point addresses that will dispatch into your BASIC program like an event. These 32 entry points addresses are returned by the function EXEVENT@. The 32 entry points dispatch through EVENTDEF entries 32 through 63. Thus, you can obtain an address to insert into the _NewWindow parameter list from EXEVENT@ and write your wContDefProc in BASIC if you like.

Task Master polling is suspended when an event is dispatched through the EVENTDEF and MENUDEF tables or through one of the EXEVENT@ entry points. All routines called by the event dispatcher are called as if a GOSUB with a line number had executed. Control is returned through the RETURN 0 statement.

The event-dispatch mechanism uses a 65816 JSL ... RTL sequence so that the external entry points provided by EXEVENT@ can call BASIC from anywhere in memory and regain control when the event-handling routine returns. The special case of RETURN 0 discards the return address of the BASIC statement interpreter and does an RTL. If the CPU stack is not properly preserved, the system will crash when the RTL uses an invalid address from the stack.

Care must be taken in writing event-handling routines in BASIC. They cannot generate BASIC errors because BASIC'S error-handling routines reset the CPU stack register, discarding the address for the above described RTL return instruction.

Opening a window file

This section discusses a special form of the OPEN statement whose use requires a through understanding of the Window Manager and its functions, as described in the *Apple IIGS Toolbox Reference* manual. Before you read this section, you should be familiar with the terms defined in Chapter xxx of that manual and understand how to use the Window Manager and QuickDraw II tool sets.

The OPEN window statement provides a means of linking a Window Manager window port to IIGS BASIC as an output file. The VARPTR( ) function returns the address of a parameter list, described in detail later in this section. When opening a window, the parameter list is an extended version of the parameters used with the Window Manager NewWindow call.

OPEN window may also be used to link a file to a QuickDraw II GrafPort through a GrafPort pointer. This variation is selected by the value of the option word in the parameter list. You must create the NewWindow parameters yourself in a structure array.

OPEN window records the file mode and its window/GrafPort pointer in the FCB, along with the optional FILTYP= parameter. If you use the optional **bufsize** parameter, a memory segment of that size is allocated and its handle is also placed in the FCB.

A window (but not a GrafPort) is closed with the CloseWindow call when the file is closed with the CLOSE # statement (or the CLOSE all variation), and the optional buffer is deallocated and the handle discarded.

After a window file has been opened, you can use the PRINT# or PRINT# USING statements to direct output text to the window at its current cursor position. IIGS BASIC does not provide for positioning the cursor like it does for the text console. You must do this yourself using the appropriate QuickDrawII calls before executing a PRINT# or PRINT# USING statement.

Each time a PRINT statement directs output to a window file, a Window Manager StartDrawing call will precede the QuickDraw II text drawing calls. IIGS BASIC uses DrawChar, DrawText, and DrawCString for output to a window file or GrafPort. When GrafPort mode is selected, a QuickDraw II SetPort call precedes the drawing functions.

The following diagram depicts the relationships between IIGS BASIC, a window port, a user tool set window driver, a data buffer, and QuickDraw II.

You must enable QuickDraw II with the GRAF INIT statement prior to opening a window file, otherwise the message

```
?TOOL SET ERROR =$0400
```

will be displayed.

The OPEN window parameter list is defined as follows:

Parameter	Type	Example	
WindowMode	WORD	I2'mode'	
Tool setNum	BYTE	I1'04'	;04= QuickDraw Tool set
DrawFuncNum	BYTE	I1'164'	;= _DrawChar in QuickDraw
Pointer	LONG	I4'0'	

The **windowmode** parameter is defined as follows:

Table 7-5

Value	Window mode	Description
$0C	Grafport draw mode	Pointer is a GrafPort address
$08	Tool set drawing	NewWindow parameters follow
$18	Tool set drawing	Insert FCB address in wRefCon
$28	Tool set drawing	Insert function 9 result in wContDefProc
$38	Tool set drawing	Insert FCB address and Func #9 result
$98	Tool set drawing	Space fill the buffer plus $18 options
$A8	Tool set drawing	Space fill the buffer plus $28 options
$B8	Tool set drawing	Space fill the buffer plus $38 options

The *DrawFuncNum* parameter is the first of four sequential function numbers used to draw text messages with QuickDraw II or a user tool set. The example shows the values for drawing text with QuickDraw II. IIGS BASIC will use *DrawFuncNum*, *DrawFuncNum*+1, *DrawFuncNum*+2, and *DrawFuncNum*+3 for DrawChar, DrawString, DrawCString, and DrawText, respectively. A user-written tool set must support all four entry points, with input parameters matching the QuickDraw II functions.

A window-driver tool set could be initialized with the address of the SANE zero page and share it with SANE (see the SANE chapter of the *Toolbox Reference* manual for restrictions). A user tool set should also be passed the address of the IIGS BASIC Task Master TaskRec as an initialization parameter. The address of the SANE zero page and the TaskRec address can be obtained by using the BASIC@ function.

You must load and initialize a window-driver tool set before opening a file that requires the tool set. The *Pointer* parameter must be zero when opening a window through the Window Manager.

A user tool set should have a tool set number from 128 through 254. The *DrawFuncNum* parameter can be any number greater than 8 and will normally be a 10. Function 9 of the user tool set (which has no inputs) should return a long result giving the address of a content update routine embedded within the tool set.

This entry point address must meet the requirements for a Task Master wContDefProc routine, as described under NewWindow in the Window Manager chapter of the *Apple IIGS Toolbox Reference* manual. Documentation on how to write a tool set is found in Appendix A of that manual.

The NewWindow parameters list is defined as follows:

Table 7-5

Parameter	Type	Description
Paramlength	WORD	Number of bytes in parameter table
wFrame	WORD	Bit vector that describes the window
wTitle	LONG	Pointer to window's title
wRefCon	LONG	Filled in by OPEN with FCB address
wZoom	RECT	Size and position of content when zoomed
wColor	LONG	Pointer to window's color table
wYOrigin	WORD	Content's vertical origin
wXOrigin	WORD	Content's horizontal origin
wDataH	WORD	Height of entire document
wDataW	WORD	Width of entire document
wMaximumH	WORD	Maximum height of content allowed by GrowWindow
wMaximumW	WORD	Maximum width of content allowed by GrowWindow
wScrollVer	WORD	Number of pixels to scroll vertically for arrows
wScrollHor	WORD	Number of pixels to scroll horizontally for arrows
wPageVer	WORD	Number of pixels to scroll vertically for page
wPageHor	WORD	Number of pixels to scroll horizontally for page
wInfoRefCon	LONG	Value passed to information-bar draw routine
wInfoHeight	WORD	Height of the information bar
wFrameDefProc	LONG	Address of standard window definition procedure
wInfoDefProc	LONG	Address of information-bar draw routine
wContDefProc	LONG	Address of content-update draw routine
wPosition	RECT	Window's starting position and size
wPlane	LONG	Window's starting plane
wStorage	LONG	Address of memory to use for window record

The FCBs internal to IIGS BASIC are divided into two halves, separated by 256 bytes. The address placed into the NewWindow wRefCon is the address of the lower half. The upper half is referenced by adding 256 to the supplied address.

Each half of the FCB contains 8 bytes of data organized as follows:

Table 7-6

Lower half	FCB1	Description
Mode	BYTE	Window mode=$08; GrafPort mode=$0C
Tool set	BYTE	The tool set number of the tool to call
Internal	BYTE	FCB master/slave control byte
Status	BYTE	FCB file status-access byte
BufrHndl	3 bytes	Handle of the data buffer memory segment (if any)
DrawFunc	BYTE	Function number of the first draw function

Table 7-7

Higher Half	FCB2	Description
FILTYP	BYTE	Value from the FILTYP= option if any else zero
Buffer Offset	WORD	Zero at open time; available to user tool set
WindowPtr	3 Bytes	Window port/GrafPort address (NewWindow result)
BufferSize	WORD	Size of the allocated buffer (if any else zero)

Both the FCB1 BufrHndl and the FCB2 WindowPtr are only 3 bytes, and you must ensure that a fourth byte of zero is created, by ANDing the upper word with #$00FF, before pushing either pointer as a parameter for a tool set call.

Using ON EXCEPTION statements

IIGS BASIC handles errors from the SANE mathematical engine as normal errors (through the ON ERR statement or by printing a message) until you use an ON EXCEPTION statement. Before you read this section, you should have a complete understanding of SANE, as described in the *Apple Numerics Manual* (published by Addison-Wesley) and have read the additional information on the SANE Tool Set in the *Apple IIGS Toolbox Reference* manual.

The ON EXCEPTION statement is a separate version of the ON ERR for the errors that occur in mathematical computations. Apple IIGS BASIC uses the SANE 65816 implementation of the IEEE Standard 754 for binary floating point arithmetic for all real and long-integer operations, or a mixed-mode operation with one real or long integer operand. OFF EXCEPTION is the default mode for IIGS BASIC, and so computational results may return an infinity or a SANE NaN result.

After an ON EXCEPTION statement has been executed, the statement list following the reserved word EXCEPTION will execute if any of the six exceptions occur during any mathematical expression evaluation. There are five SANE exceptions and one pseudo exception generated by IIGS BASIC, as follows:

Table 7-8

Bit number	Enable Value	Exception Description
1	1	An invalid operation was attempted (such as SQRT(-2))
2	2	Overflow
3	4	Underflow
4	8	Divide by zero
5	16	Inexact result
6	32	Unordered comparison (such as A<B where B is a NaN)

The EXCEPTION statement is used to select the subset of these six exceptions you want dispatched through ON EXCEPTION to your program, prior to enabling exception trapping with ON EXCEPTION.

The default exception mode of IIGS BASIC enables all exceptions except the fifth, inexact result. The sixth exception is created by the logic of the BASIC IF statement, and the exception trapping mechanism within the interpreter is called as if a SANE exception had occurred. The syntax of the EXCEPTION statement is:

```
EXCEPTION ON ubexpr
```

The unsigned byte expression (*ubexpr*) must have a value in the range of 0 through 63. The value of the expression is a bit mask used by IIGS BASIC to enable the trapping of individual exceptions. Each exception is enabled or disabled by summing the enable values shown in the above table. When an exception is disabled, its occurrence is ignored, and the expression evaluation continues as if the exception had not occurred.

The error code and error line for ON EXCEPTION are returned in ERR and ERRLIN as with the ON ERR statement. The error code returned in ERR when a SANE exception occurs may have more than one exception occur simultaneously. The *Apple Numerics Manual* defines how this can occur.

To provide complete information to your program from SANE, the value of the ERR reserved variable is set to 128 plus the masked exception byte obtained from SANE by IIGS BASIC. The exception byte values are the same as the exception enable mask bits described earlier in this section. Thus, ERR = 128 + 1, or 129, for the invalid exception and would equal 128 +1 + 16, or 145, if both the invalid and inexact exceptions occurred together. ERR can range from 129 through 191; however, in actual operation, exceptions usually occur one at a time (although inexact often occurs together with invalid).

Executing OFF EXCEPTION will restore BASIC to its default mode, in which the SANE invalid, divide by zero, not a number, underflow, and overflow errors will display messages or be trapped by ON ERR.

The following are brief descriptions of the SANE exceptions; for a complete discussion, read the *Apple Numerics Manual:*

- The invalid exception occurs for numerous reasons, but the most common one is the conversion of a real number that is too larger for a an integer format. The invalid exception also occurs when attempting to compute the square root of a negative argument or a remainder, such as x MOD y or x REMDR y where y is a zero or x is infinite.
- The underflow exception occurs when a floating-point result is both tiny and inexact.
- The divide-by-zero exception occurs when a finite nonzero number is divided by zero.

- The overflow exception occurs when a floating point destination format's largest finite number is exceeded in magnitude by the result of a computation. In other words, when the exponent range for a single- or double-precision number is too small to represent the result.
- The inexact exception occurs if the rounded result of an operation is not identical to the mathematical (exact) result. It also occurs when you convert a real number such as 10.5 to an integer.

Chapter 8

BASIC Reference

Syntax notation

The BASIC Line

Statements and Functions

ABS
ANNUITY
Arithmetic operators
Arithmetic operators
ASC
ASSIGN
ATN
AUTO
AUXID@
BASIC@
BREAK
BTN
CALL
CALL%
CATALOG
CHAIN
CHR$
CLEAR
CLOSE and CLOSE#
COMPOUND
CONT
Control-Apple-Delete
Control-Apple-Escape
Control-Reset

CONV
CONV#
CONV$
CONV&
CONV%
CONV@
COPY
COS
CREATE
DATE
DATE$
DATA
DEF FN and DEF PROC
DEL
DELETE
Digit specifications (digitspecs)
DIR
DIM
DO
EDIT
ELSE
END
Engenering specification
EOF
EOFMARK
ERASE
ERR
ERROR
ERRTXT$
EVENTDEF
EXCEPTION
EXEC
EXEVENT@
EXFN and EXFN_
EXP
EXP1
EXP2
FILE
FILTYP
FIX
Fixed-point specification
FN =
FOR ... NEXT
FRE
FREMEM

GET#
GET$
GSB.HELLO
GOSUB
GOTO
HEX$
GRAF
HLIST
HOME
HPOS and VPOS
IF ... THEN and IF ... GOTO
INDENT
INIT
INPUT
INPUT#
INPUT USING
INSTR
INT
Inteeger constants
INVERSE
INVOKE
JOYX and JOYY
KBD
LEFT$
LEN
LET
LIBFIND
LIBRARY
LIST
LISTTAB
LOAD
LOCAL
LOCK and UNLOCK
LOG
LOGB%
LOG1
LOG2
Logical expressions
Long integers
.MEMBUFR
MENUDEF
MID$
NEGATE
NEW
NORMAL

NOTRACE
OFF EOF#
ON EOF#
ON BREAK and OFF BREAK
ON ERR and OFF ERR
ON EXCEPTION and OFF EXCEPTION
ON KBD and OFF KBD
ON ... GOSUB
ON ... GOTO
ONTIMER and OFF TIMER
OPEN
OPEN window
OUTPUT#
OUTREC
PDL and PDL9
PEEK
PERFORM
PFX$
PI
POKE
POP
PREFIX
PREFIX$
PRINT
PRINT USING
PRINT#
PRINT# USING
PROGNAM$
PUT#
QUIT
R.STACK%, R.STACK@, and R.STACK&
RANDOMIZE
READ
READ#
REALS
REC
REM
RENAME
RENUMBER
REP$
RESTORE
RESUME
RETURN
RIGHT$
ROUND

RND
RUN
SAVE
SCALB
SCALE
scispec
SECONDS@
SET
SGN
SHOWDIGITS
SIN
SPACE$
SPC
SQR
STEP
STOP
STR$
Strings
SUB$
SWAP
TAB
TAN
TEN
TASKPOLL
TASKREC% and TASKREC@
TEXT
TEXTPORT
TIME$
TIME
TIMER ON and TIMER OFF
TRACE
TYP
TYPE
UBOUND
UCASE$
UIR%
UNLOCK
UNTIL
VAL
VAR
VAR$
VARPTR and VARPTR$
Variable types
VOLUMES
VPOS and HPOS

WHILE
WRITE#

Syntax notation

The syntax of a language is a body of rules that defines the various language elements and how they may be combined. There are simple elements that are combined into compound elements, which in turn can be combined into expressions and statements.

An element is defined like this:

(element to be defined)
::= (some combination of defined elements).

Any uppercase letters or punctuation marks appearing on the right side of the definition must be typed exactly as shown. Lowercase letters represent variable information that you must fill in. For example, in the definition

goto statement ::= GOTO **linenum**

the letters GOTO must be typed just as shown, followed by any legal line number.

Some definitions have two or more lines containing ::= in them. These lines are variations for a given element.

In this chapter, the following symbols are used to represent the types of elements (note that you do not type them when you are entering a program they are for purposes of describing syntax only.)

'	separates alternative elements
[	encloses optional elements
{ }	encloses repeatable elements that must occur at least once
\ \	encloses elements whose values are to be used

Other characters found in the syntax descriptions are required by BASIC.

Here is an example of how this system describes the various parts of BASIC's syntax:

house

::= roof{door}{window}[fireplace][all-electric kitchen]

A house has a roof, one or more doors, one or more windows, and may have a fireplace and an all-electric kitchen.

home

::= house'cottage'mansion

A home can be a house, cottage, or mansion.

price

::= \house\

The selling price is the value of the house.

The remainder of this chapter is a description of the functions, expressions, and statements of Apple IIGS BASIC. A concise definition of all the elements and the language syntax may be found in Appendix G, "Summary of GS BASIC."

The BASIC Lines

Apple IIGS BASIC program lines have the following format:

```
{digit} [label:] statement {: statement} {: REM comments} <Return>
```

The digit argument is the required line number, and it must be an integer in the range 1 through 65279. The optional label must begin with a letter (A through Z, a through z) and contain letters, digits, or the period. The label must be immediately followed by a colon (:) without any intervening spaces. A label may contain up to 30 characters.

Apple IIGS BASIC program lines begin with a line number, followed by an optional label, and end with a carriage return. A program line can be a maximum of 239 characters long when entered (although its internal format may be up to 286 bytes; 31 for the longest label and 255 for the line).

Program lines are entered into the current program by pressing the Return key. The carriage return does not display a character on the screen, but it is a required part of a BASIC line.

Each line must have a unique line number and, if it also has a label, a unique label. When you enter a line with a label that label, may not already be a label on a line with a different line number. For example, once a label is used on line number 3000, that label can't be used on a line with any other line number. You can remove the label from line 3000 (by editing or retyping line 3000 with no label or another label), and then use the label with some other line number. Attempting to enter a line with an existing label will display the message

```
?DUPLICATE LABEL ERROR.
```

A label may not be any reserved word. If you enter a label that is a reserved word, BASIC will display the message

```
?ILLEGAL LINE NUMBER/LABEL ERROR
```

except when the reserved word is a verb that may begin a statement. For example, if you type

```
10 NORMAL: PRINT "HI THERE"
```

the `NORMAL:` is taken as a line without a label and **not** as a line with the label `NORMAL`.

Using labels in your programs, particularly for procedures, will make your programs more readable, easier to understand, and easier to change long after you write them.

When your refer to a label in a GOTO or GOSUB statement, you must not include the colon that is entered after the label in the line. This colon allows BASIC to distinguish between a statement that begins with a variable name and one that begins with a label. For example,

```
320 TAXTOTAL = 2900.00
```

is a valid statement that is a line number followed by a word that could be a label, but is in fact a variable name. The colon tells BASIC you want the word treated as a label instead of as a variable name.

Line numbers can be preceded by any number of spaces, and there may or may not be spaces between the line number and the label or the colon and the following statement. Even if you don't enter the spaces between these elements, GS BASIC may put one between them when listing or editing the statement. You can freely use labels in your programs, but remember that long labels take up a lot more space than line numbers, which are compressed into 2 binary bytes, no matter how many digits (up to five) you have entered.

Statements and Functions

ABS

```
::= ABS(aexpr)
```

```
)PRINT ABS(345)
345
)PRINT ABS(24-363)
339
```

ABS returns the absolute value of the argument; in other words, the value of the argument if it is positive, 0 if the value is zero, and the negative of the argument value if it is negative.

ANNUITY

```
::= ANU(aexpr1,aexpr2)
```

The annuity function, ANU(rate,periods), computes the expression

```
(1 - (1 + rate)(-periods)) / rate
```

The calculation ANU(rate ,periods) is more accurate than the straightforward computation of the expression above using normal arithmetic and exponentiation operations. The annuity function is directly applicable to the computation of present value and future value of ordinary annuities. The formula for these and other useful calculations may be found in Appendix F. See also the COMPUOUND function.

Arithmetic operators

```
::= adop auop
```

This definition means an arithmetic operator is an arithmetic dyadic operator or an arithmetic unary operator. A dyadic (pronounced "di-ad-ic") requires two operands, and a unary operator requires one operand. The operands of arithmetic expressions can be single or double reals or single, double, or long integers.

There are nine arithmetic operators, two unary and seven dyadic:

Symbol	Meaning	Example	Numeric value
+	Unary plus	+5	+5
-	Unary minus	-2	-2
^	Exponentiation	2^4	16
*	Multiplication	4*6	24
/	Division	5/2	2.5
MOD	Modulo	7 MOD 5	2
DIV	Integer division	7 DIV 5	1
+	Addition	4+7	11
-	Subtraction	9-2	7

Arithmetic operators

An array is an ordered collection of variables, all of the same type. The name of the whole collection, called the array name can be any legal variable name. The last character of the name determines the type of all the variables in the array, as follows:

Example	Variable type
NAMES(2000)	Single-precision real
TOTALS#(3,2,4)	Double-precision real
COUNTR%(112)	Single integer
ADRTBL@(10,1)	Double integer
CRUZADO&(39)	Long integer
BYTS!(511,1599)	Structure

The individual variables (or elements) within an array are numbered, starting with 0. To refer to any element within an array, you specify the name of the array, followed by the number of the element enclosed in parentheses, called a subscript. For example:

```
)PRINT AR(3)
)PRINT Prices(147)
)D%(0,0)=85
```

An array may have up to 32767 elements per dimension up to the array size limit of 4096K bytes of memory. The number of dimensions is the number of subscripts needed to specify an individual element within the array. The number of dimensions and their sizes are set with the DIM (for dimension) statement.

The maximum number of dimensions is limited by the amount of data segment memory allocated via the CLEAR statement. The maximum size of a single numeric array is limited to 4 megabytes. The maximum size of a string array is limited by the 65K maximum string pool size. In addition, string arrays may not be allocated beyond the first 64K of the user data segment. (Dimension string arrays first).

The structure is a special type of array that does not have a corresponding variable type. The elements of a structure are bytes. An element of a structure will function as an unsigned single integer in a numeric expression. A numeric value assigned to a structure element must be in the range of 0 through 255, otherwise an illegal quantitty error will occur. Structures are primarily used with the GET#, PUT#, and SET statements and the VAR function.

Generally, a structure array is used by other statements as an array with an element of the size of the leftmost array dimension. For example, a structure array with

```
DIM STRUCT!(511,1599)
```

can be thought of as a one-dimensional array with 1600 elements each 512 bytes. (Ideal for reading all the blocks of a 3.5-inch disk into memory).

ASC

```
::= ASC(sexpr)
```

```
)PRINT ASC("BEEP")
)S$="Air" : PRINT ASC(s$+"horn")
```

ASC returns the decimal ASCII code corresponding to the first character of the given string expression. If the string expression value is a null string, then the value -1 is returned.

ASSIGN

```
::= ASSIGN chardevicename,aexpr[,AUTO]
```

ASSIGN associates a character device name with a slot or port number. GS BASIC defines six standard character device names, which may be changed with ASSIGN. A **chardevicename** is a filename that begins with a period, followed by a letter (A through Z, a through z) followed by one or more letters or digits (not including a comma).

The optional AUTO argument indicates that BASIC should also send a line-feed character after each carriage return sent to the device. After a new **chardevicename** is defined with ASSIGN, the new name can be used in the OPEN statement to access a device for input or output.

A limit of 12 **chardevicenames** may be defined with ASSIGN (including the six default names). A value of 1 through 7 defines the slot number of the character device. A value of 0 defines a null device, and the value -1 deletes a **chardevicename** from the table. Any other negative number will cause an illegal quantity error.

The six standard character device names are:

Device name	Slot	Auto-LF	Description
.CONSOLE	3	OFF	C3COUT1
.PRINTER	1	ON	
.MODEM	2	OFF	
.MEMBUFR	-	OFF	pseudo-device (255-byte buffer)
.NETPTR1	7	ON	(AppleTalk® printer driver)
.NULL	0	OFF	(a bit bucket, read=CR)

ATN

```
::= ATN(aexpr)

)PRINT ATN(.3456)
.33275
)
```

Returns the arc tangent, in radians, of the given argument. The value returned represents an angle in the range -pi/2 to +pi/2 radians.

AUTO

```
::= AUTO [linenum [,- increment]]
```

AUTO is a submode of the EDIT command that automatically generates line numbers for new lines to be entered into the current program. The **linenum** is the first line number presented after entering edit mode. The increment is the amount added to **linenum** to generate the next line number.

If the **linenum** or the increment are not given, the value 10 is used. Entering an increment larger than 1000 will be ignored, and the value 10 will be used instead.

If **linenum** is the line number or label of an existing line, the command is executed as if the command EDIT **linenum** - **linenum+increment** was entered. If a label is used and that label does not exist, the message

```
LINE NUMBER/LABLE ERROR
```

will be displayed.

AUXID@

```
::= AUXID@
```

AUXID@ is a reserved variable that is set each time an OPEN, LOAD, SAVE, RUN, CHAIN, or COPY statement or the FILE? function is executed. It returns the value of the catalog Subtype field as an unsigned double integer for the last file referenced by any of these commands.

BASIC@

```
::= BASIC@(ubexpr)
```

The BASIC@ function is provided to allow easy access to the addresses of certain memory resouces that are allocated by GS BASIC. The unsigned byte expression must be a number from 0 through 255, and the returned result is always a positive double integer that is an address of some data structure internal to GS BASIC.

The argument selects one entry in a table maintained by BASIC during execution. The details of the BASIC@ function are described in Appendix D, "Interpreter Data Structures."

BREAK

```
::= BREAK ON
::= BREAK OFF
```

During program execution, except when reading from the keyboard for INPUT, GS BASIC monitors the keyboard for the attention keypress, a Control-C. When this key combination is pressed, BASIC discards any characters in the typeahead buffer, stops the program after the current program statement is completed, and displays the message

```
PROGRAM INTERRUPTED IN linenum
```

Control-C monitoring can be suppressed by BREAK OFF and re-enabled by BREAK ON. If Control-C is pressed while BREAK is off, it is treated like any other character and entered into the typeahead buffer. BREAK OFF also disables the recognition of Control-C during the INPUT statement.

BREAK OFF is the startup mode until GS BASIC displays the command line prompt for the first time. The GSB.HELLO program, if present, is run with BREAK OFF in effect until it executes an END or STOP statement or until the program specifically reenables BREAK ON.

BTN

```
::= BTN(0 <=ubexpr <= 2)
```

BTN returns the state of the three sense inputs ($E0C061, 62, and 63) as 0 or 1. The various devices that control the state of these inputs include paddle or joystick, buttons and the Apple and Option keys.

CALL

```
::= CALL libname [( param [{,param}] )]
::= _libname [( param [{,param}] )]

)CALL PAINTRECT(VARPTR(RECT%(0)))

)_PAINTRRECT(VARPTR(RRECT%(0)),50,20)
```

CALL executes a named procedure in an Apple IIGS tool set. Tools and/or their interface definitions are loaded by the LIBRARY statement and must be properly initialized by your BASIC program. The function number, tool number, parameter requirements: result size, and result type, if any, are extracted from the interface definition library by searching the current dictionary for **libname.** The interface definitions for a tool set are loaded by the LIBRARY statement from a special TDF (FILTYP = TDF).

The reserved word CALL can be replaced by the shorthand call character, the underscore_. The TDF for the tool set must have been loaded into the LIBRARY dictionary with the LIBRARY statement prior to executing a CALL **libname:** otherwise an

```
UNDEF'D PROC/FUNCTION ERROR
```

message will be displayed.

The dictionary entry indicates if any parameters are required by the tool function (and their order and types). The parameter list must contain the proper number, order, and types of arguments within parentheses following the **libname** if any parameters are required. The parameters are pushed on the CPU stack in order from left to right, tool set and the proper tool set function is called by the Tool Locator.

Any returned results are removed from the CPU stack, and the first 16 words (32 bytes) are stored in the return stack buffer. The contents of the return stack buffer may be accessed by the R.STACK functions.

The Apple IIGS Tool Locator interface is used to dispatch the function call. Apple IIGS BASIC is supplied with a set of TDF for most of the tool sets available at product release. The **libnames** for the functions will match, as far as possible, the function names used in the *Apple IIGS Toolbox Reference manuals* for the tool set. More information on the format of the TDF is found in Appendix H.

Don't attempt to use CALL without complete knowledge of a tool set. GS BASIC correctly initiates QuickDraw II when the GRAF INIT command is executed, and it also does the startup of the Sound tool set (but not the NOTESYN or NOTESEQ tool sets). You may obtain the addresses of some preallocated memory resources that are useful for initializing certain tool sets, with the BASIC@ function.

To pass real or integer numbers or the values of variables, include them in the argument list as an expression (for an explanation of expressions, see "Expressions" in Chapter 2). If the type of the numeric argument or expression you use does not match the type of the argument required by the tool set function, CALL attempts to convert the result to the proper type as if you had used the CONVx( ) function for the type of the argument. Argument conversion does not include string-to-numeric or numeric-to-string conversions. The message

```
?ARGUMENT TYPE MISMATCH ERROR
```

will be displayed in these cases.

You must also use the correct number of arguments when calling a tool set function, otherwise, an argument count error will occur.

- *NOTE:* The binary format of real numbers are those defined by the SANE tool set. If an expression is used for a parameter, the expression evaluation may create an extended-precision real result, which will be converted to the type required by the tool set function. This conversion may cause an overflow error if the result of the expression is a number too large for the type of parameter required by that argument.

To pass the address of a numeric variable, use the VARPTR( ) function. There is no means of passing the address of an expression.

If the tool set interface definition indicates that the argument for a tool set function should be a counted string (often referred to as a Pascal string), the CALL statement will convert a BASIC string, or string expression result, into a counted string. A counted string is a count byte followed by the characters. CALL passes the address of the count byte to the function instead of the address of the BASIC string descriptor.

WARNING:

The counted string conversion will only pass strings up to 254 characters long. Attempting to pass a string with 255 characters will cause the string too long error to occur.

To pass the address of the string's first character (without a count byte) use the VARPTR$( ) function.

CALL%

```
::= CALL% ubexpr1,ubexpr2,ubexpr3 [( param{,param} )
```

```
)CALL%  84,4,0(@RECT%(0)
)CALL%  94,4,0(@RRECT%(0),%50,%20)
```

CALL% executes a specified procedure of a specified tool set properly initiated previously by your program. The tool function number is given by **ubexpr1,** the tool set number is given by **ubexpr2,** and the size of the result (in words), if any, returned on the stack is given by **ubexpr3.** Any returned results are removed from the CPU stack, and the first 16 words (32 bytes) are stored in the return stack buffer. The contents of the return stack buffer may be accessed through the R.STACK functions.

All 3 unsigned byte expressions (**ubexpr**) must be in the range of 0 through 255 or an illegal quantity error will occur. The Apple IIGS Tool Locator is used to dispatch the function call; refer to the proper *Apple IIGS Toolbox Technical Reference* manual chapter for the tool set you are using.

If an argument list is present (enclosed in parentheses after the **ubexpr3**), each argument is evaluated and pushed onto the stack (from left to right) before execution of the procedure. Correct operation of toolbox procedures requires that proper parameters be pushed on the stack in the correct order. Failure to do this will probably crash the system.

To pass real or integer numbers or the values of variables, just include them in the argument list. If an expression is used, the result may be an extended-format real number, in which case a 10-byte parameter is passed to the function (which is likely to crash the system unless the function is expecting a SANE extended-format number as the parameter).

To pass the address of a numeric variable use the VARPTR( ) function. You may not pass the address of an expression. CALL% does automatic conversion to a counted string like CALL (described above) for any string parameter. CALL% does not use the LIBRARY dictionary interface definitions, and so the number and types of the parameters are not checked, nor are automatic numeric conversions performed, but CALL% is faster as a result.

When you want to use an expression as an argument, you can force the result to be the correct type by using a conversion function such as CONV%( ) or CONV@( ), I.E. You must put any required conversions explicitly in your program. Integer constants will be passed as integer, double integer, or long integer (2, 4, or 8 bytes), based on the range of the value. Thus, if you want a double integer zero, you must use CONV@(0).

CATALOG

```
::= CATALOG [diskname · dirpath · sexpr]        immediate mode
::= CATALOG [sexpr]                             deferred mode
::= CAT [diskname · dirpath · sexpr]            immediate mode
::= CAT [sexpr]                                 deferred mode
```

```
)CAT .D2
)CAT
)CATALOG /Apple1
)CATALOG /Apple1/Applekind
```

CATALOG displays an 80-column listing of a root directory or subdirectory specified by the directory path following the reserved word CATALOG. CAT displays the left 40 columns of the 80-column listing. If the specified pathname is a given volume name, then the names of all files in the given root directory are displayed on the screen, and the names of any subdirectories of the root directory are also displayed.

If no pathname is given, the pathname contained in PREFIX$ is assumed.

If OUTPUT# is set to anything other than 0, the directory listing will be sent to the specified OUTPUT file, not to the screen.

CHAIN

```
::= CHAIN pathname [,linenum·label]
```

```
)CHAIN Lightning,Strikes
```

CHAIN automatically loads and runs a specified program, without clearing the values of the variables left over from the previous program or closing any files the previous program had left open. This allows variable values used in one program to be used in another. The program begins execution at the line given by the optional line number or label. The pathname of the program to chain must follow the reserved word CHAIN. If the chained program assigns dimensions to an array that was defined in the previous program, the message

?DUPLICATE DEFINITION ERROR

is displayed.

CHR$

```
::= CHR$(aexpr)
```

```
)PRINT CHR$(66.8)
)R$="68" : PRINT CHR$(VAL(R$))
```

CHR$ returns the ASCII characters corresponding to the value of the arithmetic argument, which must be in the range of 0 through 255, or an illegal quantity error occurs. (See Appendix A, "ASCII Character Codes".)

CLEAR

```
::= CLEAR
::= CLEAR iexpr
::= CLEAR INVOKE
::= CLEAR LIBRARY
```

CLEAR, with no parameters, sets all numeric variables to zero, all strings variables to null strings, clears all BASIC pointers and stacks, and closes all open disk files except a file being executed.

CLEAR **iexpr** attempts to change the data segment size while preserving all variables, strings, arrays, library dictionary, and open files. The amount of memory allocated for arrays, variables, and string data is adjusted to the size given (rounded up to the nearest multiple of 256) or the minimum amount currently required, whichever is greater. The minimum required memory includes the sum of all arrays, variables, local variables, string data, and invoke records, plus a minimum of 2048 bytes of free space for string operations.

The **iexpr** may not be set smaller than 8192 bytes or the minimum requirement.

Additional memory segments are allocated for the BASIC program, the library dictionary tables, plus one for each invoked module and each file record buffer. (See the LIBRARY and INVOKE statements in this chapter and Chapter 7, "Advanced Topics.")

CLEAR LIBRARY and CLEAR INVOKE delete all the dictionaries and code segments that have been loaded with LIBRARY and INVOKE statements, respectively. In addition, executing CLEAR LIBRARY and CLEAR INVOKE will shrink the library memory segment to its minimum size (512 bytes). INVOKE and LIBRARY both insert interface definitions into the library segment. CLEAR LIBRARY or CLEAR INVOKE used separately will only shrink the library segment to minimum when there are no definitions for the other purpose.

CLOSE and CLOSE#

```
::= CLOSE[# filenum]

)1000 CLOSE

)Strikes: 3200 CLOSE# 1
```

Before ending program execution, all open files should be closed with either a CLOSE# or CLOSE statement. Any files closed during program execution must be reopened before they can be accessed again. Each time a file is opened, even if it was used earlier in the same program, BASIC assumes that the file has not been opened before during the current execution of the program.

CLOSE# closes the file whose file number is equal to the arithmetic expression that follows CLOSE#.

CLOSE closes all files that are open when the statement is executed. All open files are also closed by a LOAD, CLEAR, NEW, or RUN statement. The CHAIN statement does not close any files.

COMPOUND

```
::= COMPI(aexpr,aexpr)
```

The compound interest function, COMPI(rate,periods), computes the expression:

```
(1 + rate) periods
```

When the rate is small, COMPI(rate ,period) gives a more accurate result for the computation than does the straightforward computation of $(1+r)^P$ by addition and exponentiation. COMPI is directly applicable to computation of present and future values and the formula for these are found in Appendix F. See also the ANNUITY function.

CONT

```
::= CONT
```

CONT resumes execution of a program that has been halted by a STOP or END statement or a Control-C at the statement immediately following the one at which execution was suspended. CONT can only be used as a command, not as a statement in a program.

CONT does not clear the program or reset the variables in memory, and there are no options associated with it.

A program halted by an error may be continued. BASIC will attempt to continue execution starting with the statement in which the error occurred. An error made in immediate execution will not prevent a program from being continued.

A program that has had any of its statements altered, or any new statements added, may not be continued. If you try, the

```
?CAN'T CONTINUE ERROR
```

message will be displayed. The values of variables in a program can be changed using assignment statements in immediate execution while the program is stopped.

GS BASIC suspends a number of activities during immediate mode that are restarted when execution is resumed with CONT. The ON TIMER interval counter is frozen and Task Master polling is suppressed during immediate mode. The status of ON TIMER, ON KBD, and TASKPOLL are preserved and restarted when CONT is used.

Control-Apple-Delete

Pressing the Control-Apple-Delete key combination clears the keyboard micro and typeahead input buffers.

Control-Apple-Esc

Pressing the Control-Apple-Esc key combination will enter the Apple IIGS Control Panel program, which may cause a change of screen modes. For more information about the Control Panel, refer to the *Apple IIGS Owner's Guide*.

Control-Reset

Pressing the Reset button while holding down the CONTROL key reboots your Apple IIGS, just as if you had switched the computer off and back on. Anything stored in memory is lost (including your program and GS BASIC).

CONV

```
::= CONV(expr)

)G&=234234 : H&=523523 : PRINT CONV(H&-G&)
289289
)
```

CONV evaluates the argument, and returns a single-precision real value. The value may be assigned to an integer variable. If CONV is used with a string expression, the effect is the same as with the VAL function.

CONV#

```
::= CONV#(expr)

)G&=234234 : H&=523523 : PRINT CONV#(H&-G&)
289289
)
```

CONV# evaluates the argument and returns a double-precision real value. The value may be assigned to an integer variable. If CONV# is used with a string expression, the effect is the same as with the VAL function.

CONV$

```
::= CONV$(expr)
```

```
)D%=345 : A%=453 : PRINT "a"+CONV$(D%*A%)+"z"
a156285z
)
```

CONV$ evaluates the given expression and returns a string value.

CONV&

```
::= CONV&(expr)
```

```
)PRINT CONV&(2178-7954)
-5776
)PRINT CONV&("62578942179.85")
62578942179
)
```

CONV& evaluates the given argument and returns a long-integer value.

If the argument is a string, the effect is the same as using VAL followed by CONV& (see the chapter STRINGS AND STRING FUNCTIONS for an explanation of the VAL function). The value returned must be within the range of 9,223,372,036,854,775,807 through -9,223,372,036,854,775,807, or an overflow error will occur. The result may be assigned to a single- or double-real variable, although significance will be lost in the conversion. The largest negative number, -9,223,372,036,854,775,808, is used to represent the SANE NaN result.

CONV%

```
::= CONV%(expr)
```

```
)PRINT CONV%(423.94)
424
)A&=7656 : B&=364 : PRINT CONV%(A&/B&)
21
)
```

CONV% evaluates the argument and returns an single integer value, rounding off to the nearest whole number. The value returned must be within the range -32768 to 32767, or an overflow error results.

CONV@

```
::= CONV@(expr)
```

```
)PRINT  CONV@(765423.94)
765424
)A=7656.43  :  B=364.11  :  PRINT  CONV@(A/B)
21
)
```

CONV@ evaluates the argument and returns an double-integer value, rounding off to the nearest whole number. The value returned must be within the range -2,147,483,648 to 2,147,486,647, or an overflow error occurs. A result of less than .5 will be rounded down to zero.

COPY

```
::= COPY filename1,filename2[,linenum label]
```

COPY is used to copy any disk file within or between disk volumes. Both **filename1** and **filename2** must be enclosed in quotation marks in deferred mode or be valid string expressions. Wildcard filenames and device filenames are not allowed; COPY only copies one existing disk file to a new volume or different subdirectory within a volume. COPY preserves the file type and subtype of the original file.

COPY will return an out of memory error if less than 1280 bytes of free data segment memory are available. COPY will use up to 63.5K of memory as a buffer for the copy operation. Files larger than 63.5K will be copied in 63.5K increments (assuming 63.5K is available). When copying between two disk volumes, both volumes must be mounted during the entire copy operation, unless the optional **linenum** or **label** is present. If the output filename already exists, a duplicate filename error will result.

The optional **linenum** must reference a subroutine that ends with a RESUME COPY statement. The subroutine will be called by COPY before each reading of the input file and before each write action to the output file. The initial call occurs just before COPY trys to locate the input file, the second call occurs before it trys to verify that the output file doesn't exist, the third call occurs before the first read, and the fourth occurs before the first write. The subroutine should prompt the user to switch disk volumes for a single drive copy. The optional **linenum** parameter cannot be used in immediate mode.

COS

```
::= COS(aexpr)
```

```
)PRINT COS(1.571)
-2.03673E-04
)
```

COS returns the cosine of an angle given in radians.

CREATE

```
::= CREATE pathname, [FILTYP=DIR ' TXT ' SRC ' BDF ' ubexpr] [,aexpr]

)CREATE "/Pies/Applepie", FILTYP=TXT
)CREATE Attache, FILTYP=172, 4212
```

CREATE is used to make subdirectories, text files, source files, data files, or any other file type. You must specify the name of the file you want to create by following the reserved word CREATE with the new pathname. The optional FILTYP= parameter may follow the pathname separated by a comma. TXT specifies that a text file be created; BDF specifies that a BASIC data file be created; and DIR specifies a subdirectory. If the FILTYP= option is not used, a TXT file is created. Three synonyms are supported for the FILTYP= option: TEXT may be used for TXT files, DATA may be used for BDF files, and CAT may be used for DIR files.

A file's **Subtype** may be specified by appending an arithmetic expression to the CREATE argument list. The subtype is required only for random-access data files, and it specifies the logical record size. GS BASIC requires this record size be in the range of 3 through 32,767. For other file types, the range of 0 through 65,535 is allowed, and the meaning of the subtype varies according to its file type. If no subtype argument is given, the subtype defaults to 0. A subtype of zero is always used for DIR files, regardless of the subtype given.

Any arbitrary file type can be created with the FILTYP=**ubexpr** option. The unsigned byte expression (**ubexpr**) must result in a number in the range of 0 through 255; otherwise, an illegal quantity error results. A partial list of the most common file types is found in Appendix J. You must understand how the subtype is used by other application programs for their specific file types if you are creating such a file.

An attempt to create an already existing file generates a duplicate file error.

DATE

```
::=DATE(ubexpr)
```

The DATE function is used to read the Apple IIGS clock date fields as numbers rather than as the variant format string returned by DATE$. The **ubexpr** must be in the range of 0 through 4. The DATE function must be called with a zero argument to actually update the values returned for the other arguments to the current date. The result returned for argument zero is the year less 1900, for example 87 rather than 1987.

The requirement that the function zero be called first protects you from having the date, month, or year change between calls for the other results. This problem is commonly known as "the clock rollover" problem. If DATE(0) is not called immediately prior to using DATE with the other parameters, the function results will reflect the date at the time of the previous DATE(0) call. You should not call DATE(0) a second time until you have retrieved all the other results into your variables.

Function	Result
DATE(0)	Year–1900, (87 is returned for 1987)
DATE(1)	Year
DATE(2)	Month (1 through 12)
DATE(3)	Day of month (1 through 31)
DATE(4)	Day of week (1 through 7, Sunday = 1)

DATA

```
::=DATA [literal`string`real`integer][{, [literal`string`real`integer]}]

)1158 DATA "Panjandrum",1.41421,Deficit&
```

DATA creates a list of elements that can be used by a READ statement.

DATE$

```
::=DATE$
::=DATE$ ubexpr1,ubexpr2,ubexpr3
```

DATE$ is both a reserved variable (first form) that returns the current date as a string, and a statement (second form) that sets the Apple IIGS clock date to the year given by **ubexpr1,** the month given by **ubexpr2,** and the day of the month given by **ubexpr3.** You must be sure that the parameters fall into the following ranges: year 0 through 255 = year–1900, month 1 through 12, and day 1 through 31. The current time setting is not disturbed by changing the date.

The string format for the first form is determined by the date format setting in the Control Panel.

DEF FN and DEF PROC

::= DEF FN **name**[%`@`&`#`$](**var**{,**var**}) = **expr**
or
::= DEF FN **name**[%`@`&`#](**var**{,**var**}): [**statement list**]
...
FNname = expression
...
END FN **functionname**

::= DEF PROC **name**[(**var**{,**var**})]: [**statement list**]
...
END PROC [**procname**]

```
)10 DEF FN Circumf(X) = X*2*PI
)20 DEF FN Sworded#(C#) = INT(RND(3)*100)
)30 DEF FN M5BY7.MAT(DED) = DED*LOG(33)-ABS(F%)
```

The DEF statement allows you to define functions and procedures for use in your programs. DEF FN allows you to define single-expression functions or multiline functions. Multiline functions can only be referenced in a LET statement expression. Single-line functions can return string values, whereas multiline functions can only return numeric values.

When you execute your program, BASIC searches through all the lines of your program for DEF FN and DEF PROC statements and builds a special entry in the variable table for each function or procedure. This DEF scan validates the DEF statement and the corresponding END statement and may generate error messages if it finds incorrect syntax.

❖ *Note:* The DEF statement must always be the first statement in a program line. If you embed a DEF statement later, the DEF scan will not find it, and the message

```
?UNDEF'D PROC/FUNCTION ERROR
```

will be displayed when your program references the function or procedure.

DEF FN allows you to define multiline numeric functions to be used in your programs. The function's arguments must be simple variables. Each argument becomes a local variable in the temporary local variable table. You may create additional local variables with the LOCAL statement in multiline functions. All function definitions must have at least one argument, but procedures need not have any.

Procedures are generally faster than GOSUB routines because the program is searched once for the location of the procedure during the DEF scan instead of every time it is called.

Chapter 7, "Advanced Topics," provides a more complete discussion of functions and procedures.

DEL

```
::= DEL linenum1`label1 [ ,`- linenum2`label2 ]
```

DEL deletes lines from the program stored in memory. You can specify either a single line or a range of lines to be deleted. This command can only be used in immediate mode; it cannot be a statement in a program.

```
)DEL 7
)DEL 73,193
)DEL PAST-FUTURE
```

Each of the examples above will delete all existing lines of the program currently in memory within the specified range (including the single line).

DELETE

```
::= DELETE pathname

)DELETE /Tree/Banana
```

The DELETE statement is used to remove the subdirectory or file specified as its argument. A subdirectory can be removed only if all files in that subdirectory have been deleted.

Even if all files in a root directory have been removed, you cannot remove the root directory.

A number of errors can occur with improper arguments appended to a DELETE statement. They are summarized below.

Error message	Cause
?VOLUME NOT FOUND ERROR	Volume name given does not exist.
?PATH NOT FOUND ERROR	Subdirectory does not exist.
?FILE NOT FOUND ERROR	Local file name does not exist.
?FILE LOCKED ERROR	Subdirectory contains files, or specified file is locked.
?WRITE PROTECTED ERROR	Diskette is write-protected.
?FILE BUSY ERROR	The file is already open.

Digit specifications (digitspecs)

A # reserves one numeric digit position. Leading zeros (if present) are replaced with spaces.

A Z reserves one numeric digit position, just like a #, except that leading zeros are printed.

An & character reserves one position for a numeric digit or comma. Commas are inserted after every third digit left of the decimal point. Commas are included in the character count, and leading zeros are replaced with spaces. At least five digit positions must be reserved to the left of the decimal point when using &.

DIR

```
::= DIR [diskname : dirpath [/wildcard[,typlist]]:sexpr]
::= DIR [wildcard [,typlist]                    immediate mode
::= DIR [sexpr]                                 deferred mode
```

```
)DIR .D2
)DIR
)DIR /Apple1
)DIR =,GSB,TDF
)DIR =.TXT,TXT
)DIR AB=#=
)DIR =,-GSB,TDF,TXT,BIN
1000 DIR "/Apple1/Applekind"
```

DIR displays an 80-column listing of a root directory or subdirectory specified by the directory path following the reserved word DIR. If the specified pathname is a given volume name, then the names of all files in the given root directory are displayed on the screen and the names of any subdirectories of the root directory are also displayed.

One or two MouseText characters will precede the display of the entries from the directory to create a visual key (a pseudo-icon) for the file types commonly used with IIGS BASIC. A MouseText icon will be assigned for BASIC program files (GSB), data files (BDF), TDF files, TXT files, and DOC files, as well as SYS, O.S, and BIN files.

The wildcard field is a special form of a filename defined solely within the DIR command. The wildcard filename defines a pattern used for selecting a subset of the entries within a directory or subdirectory. Three special characters have wildcard meanings, and all other characters must be matched exactly *including spaces,* except that uppercase and lowercase letters are treated as the same.

The two simple wildcard characters are the number sign (#) and the dash (-). The number sign character matches any one-digit character, and the dash character matches any one character (including digits). For example, the wildcard pattern P# will match all two-character filenames that begin with p or P and end with a digit.

The other wildcard character is the equal sign character (=). This will match any number of characters of the directory entry name. Thus, a wildcard pattern of just an equal sign will match all entries in the directory. Normally, the = character is used before or after other characters to create a pattern with some fixed and some wildcard characters. For example, the pattern =.TXT will select all the filenames that begin with any characters and end with the four characters .TXT. The equal sign character cannot be followed immediately by another equal sign or dash character in a wildcard pattern.

The equal sign character can be used more than once in the same pattern, with the above limitation. DIR allows for up to 32 characters in directory entry names and patterns. The # and - characters can be used as often as needed, and they can be used sequentially.

The optional **typlist** is defined as follows:

```
::= [-]TYP{,TYP}
```

The TYP fields are the three-letter file type abbreviations used by DIR and CATALOG in their TYPE field output display. When a **typlist** is given following a wildcard pattern, only those directory entries with file types matching the **typlist** are displayed by DIR. The optional leading minus sign inverts the meaning of the entire list of file types and indicates that all file types except those given in the list are to be displayed. In deferred mode, a TYP field may be a numeric expression with a value from 0 through 255.

If no pathname is given, the pathname contained in PREFIX$ (prefix 0) is assumed, along with a default wildcard pattern of a single equal sign and all file types.

If OUTPUT# is set to anything other than 0, the directory listing will be sent to the specified OUTPUT file instead of to the screen.

DIM

```
::= DIM array name (subscript{,subscript})
                          {,array name(subscript{,subscript})}
```

```
)DIM MIND%(7,2,3)
)DIM BLOCKS!(511,1599), Bulbs@(2,45), Lanterns&(9,0,8), LY(16)
```

You can allocate space for one or more arrays in your program with a DIM (for dimension) statement. The maximum size for a subscript is 32767, and the maximum total size for a single numeric array is 4096K (4,194,000 bytes).

If you assign a value to an array before defining it with a DIM statement, BASIC automatically creates an array having 11 elements per dimension, with subscripts numbered from 0 through 10. If the statement

```
)PRINT D&(18,LOOP5)
```

is executed before the array D& is defined, zero is printed but no array is created.

If the value of a subscript refers to either a nonexistent dimension or a nonexistent element (one that is greater than the highest numbered element in a given dimension), a bad subscript error occurs.

A special type of array, called a structure, can be defined with the DIM statement. There is no corresponding simple variable for this type of array. A structure array uses the exclamation point (!) as its type character. A detailed description of structures can be found under "Arrays" earlier in this chapter.

DO

```
::= DO
```

The DO statement defines the beginning of a DO ... UNTIL loop or a DO ... WHILE ... UNTIL loop. (See the WHILE and UNTIL statement descriptions later in this chapter.)

EDIT and EDIT TO#

```
::= EDIT linenum1 'label1 [,'- linenum2 'label2]

::= EDIT TO# filenum
```

The EDIT command is used to edit the statements of your BASIC program. It will search the program for the line with **linenum1** or **label1** and display that line or the next line in the program for editing. Edit mode divides the text screen into two textports. The bottom four lines of the screen are used for editing the current line, and the remainder of the screen functions as the normal scrolling display.

Editing is done using the Apple IIGS UIR editor. Pressing the Return key indicates the line is complete. The Escape key will terminate edit mode and restore the screen to a single 24 by 80 scrolling textport. The following UIR edit commands are supported:

Control-D and Del	Deletes the character to the left of the cursor
Control-E	Selects insert versus replace mode and cursor
Control-F	Deletes the character under the cursor
Control-H(<--)	Moves the cursor one character to the left
Control-U(-->)	Moves the cursor one character to the right
Control-X	Deletes all the characters in the line
Control-Y	Deletes all the character from the cursor to the end of the line
Control-Z	Restores the line to its original value

The EDIT and AUTO commands are described in more detail in the section titled "Editing Your Programs" in Chapter 1.

EDIT TO # **filenum** is an additional mode for the EDIT command that will write text lines to the open file indicated by the file number **filenum**, one line at a time. No line number or program line will appear in the edit window, just the insert cursor. The screen will scroll up two lines before the edit window appears.

The open file may be a ProDOS TXT or SRC file, opened either FOR OUTPUT or FOR APPEND, or a character device file such as PRINTER. Each line is sent to the file and echoed to the main window when the Return key is pressed. Since TXT and SRC files always have a 512-byte ProDOS buffer, the text will not be written to disk until 512 bytes have been entered. You must open the file yourself, to assign a **filenum** to it, and close it to ensure that the last lines typed are flushed to the disk file. As with the EDIT command, the edit mode is exited by pressing the Escape key.

EDIT TO will not write to files with file types other than TXT or SRC or write to files opened only FOR INPUT. When EDIT TO mode is used the immediate mode versus deferred mode distinction normally present during EDIT does not operate. Everything you type, even if it begins with digits, is written to the file and echoed to the

screen. If what you type happens to be a valid BASIC program line, it is still treated as ordinary text and not checked for syntax or tokenized.

EDIT and EDIT TO# force 80-column mode, normal text, with MouseText off.

ELSE

```
::= ELSE linenum'label'statement  list
```

The ELSE statement can be used as a continuation line in a multiline IF ... THEN ... ELSE conditional logic statement. If an ELSE statement occurs alone or is branched to directly, it behaves like a REM statement, skipping everything on the line. (See the IF ... THEN statement description later in this chapter.)

END

```
::= END
::= END FN name
::= END PROC [name]
```

END without any parameters is the same as STOP, except that no message is displayed. END FN **name** and END PROC [**name**] are used in conjunction with DEF FN and DEF PROC, respectively. See the DEF statement description above for details.)

Engenering specification

```
::= [+'-] engrpart [fracpart] exp
```

```
)PRINT USING "+3#.4#4E"; 1729
+ 1.7290E+03
)PRINT USING "+3Z.4Z3E"; 1729
+01.729E+3
```

The engineering specification (engrspec) is closely related to the scientific notation specification. It forces the exponent's value to be a multiple of 3, and has a maximum of three-digit positions to the left of the decimal point.

Either #'s or Z's can be used to indicate digit positions, and their choice is significant only to the left of the decimal point; # replaces leading zeros with spaces, and Z prints leading zeros.

EOF

BASIC assigns the file reference number of the file causing an EOF error to the reserved variable EOF. You can then check the reserved variable EOF to determine the affected file.

When you use the reserved variable EOF in an ON ... GOTO or ON ... GOSUB statement, you must enclose EOF in parentheses. For example:

```
)ON (EOF) GOTO 100,200,300
```

EOFMARK

```
::= EOFMARK(filenum)
```

EOFMARK returns the current end-of-file mark for the specified open file. The value returned is in the range of a positive double integer.

ERASE

```
::= ERASE variable-name'array-name() {,variable name'array name()}
```

ERASE deletes the variable or array and frees all the memory space it occupied. Any data in the array or variable is immediately lost and cannot be recovered. The array names must be followed by two parentheses, as shown.

The freed memory will be used for new variables, arrays, or string data. For the memory to be used for file buffers, invokable modules, or the library dictionary, you must first use the CLEAR statement must be used to reduce the size of the data segment.

ERR

When BASIC encounters an error, it assigns the reserved variable ERR a code number corresponding to the type of the detected error. You can then refer to the reserved variable ERR to determine what kind of error occurred. For a list of these codes and the corresponding error messages, see Appendix B, "Error Messages."

ERROR

```
::= ERROR ubexpr
```

The ERROR statement generates a user-defined error code, which can be trapped by the ON ERR statement. The **ubexpr** may be any number from 1 through 255. If the number equals a defined BASIC error number that error message will be displayed; otherwise, the message

```
USER PROGRAM ERROR =nnn
```

will be displayed.

ERRTXT$

```
::= ERRTXT$(ubexpr)
```

The ERRTXT$ function returns a string, which is the text of the error message for the error number given by **ubexpr.** If **ubexpr** is not a BASIC error number, a null string is returned.

EVENTDEF

```
::= EVENTDEF index{,linenum'label}
```

EVENTDEF defines the beginning line numbers of the event-handling routines that will be called after TASKPOLL ON is executed. TASKPOLL ON activates the polling for Task Master events. IIGS BASIC has an internal table of 64 events, 27 of which may be returned by Task Master. The internal table is indexed using the event code number returned by Task Master. If the entry in the table is zero, the event is ignored (except for event 17).

EVENTDEF defines the line number to dispatch control to when a Task Master event is returned. The index parameter is a number from 0 to 63 that defines which event you want to handle in your program, and the **linnum** is the event-handling routine. The Task Master documentation under the Window Manager section of the *Apple IIGS Toolbox Reference* manual defines the meaning of the various event numbers from 1 through 27.

The routines referenced by EVENTDEF must end with a RETURN 0 (zero) statement. This special form of the RETURN statement can only be used as the last statement in an event-handling routine for Task Master events.

EVENTDEF can define sequential events by following the index with multiple line numbers, separated by commas. If you want to skip an event, use a **linnum** of zero. Normally, using Task Master only requires defining a few of the 27 possible events; see Using Task Master in the *Toolbox Reference* manual.

Event 17 is a special case related to the menu events from Task Master. As long as event 17 is left a zero, IIGS BASIC assumes that you have defined the menu-item-handling routines with the MENUDEF statement and directs the individual item-select events to the event-handling routines. (See the MENUDEF description in this chapter.)

EXCEPTION

```
::= EXCEPTION ON ubexpr
::= EXCEPTION OFF
::= EXCEPTION 0
```

IIGS BASIC implements floating-point arithmetic operations with the SANE mathematical routines (tool set) and provides the programmer with control over the exceptions generated by the tool set. There are three modes available in handling these exceptions that can be selected by the EXCEPTION statement.

The default mode is selected with EXCEPTION OFF. Unless you have read the *Apple Numerics Manual* and have a complete understanding of what exceptions are and how to use them, you need not change the exception-handling mode. In the default mode, IIGS BASIC returns the standard error messages for the important mathematical calculation exceptions and ignores the unimportant exceptions. The details of exception handling in the default and other modes are provided in Appendix K.

EXCEPTION 0 (zero) is used to disable all SANE exceptions and will cause all exceptions to be ignored and pass through NaN's to the expression results and into the real variables.

EXCEPTION ON is used to enable exception trapping in your program of a specific type beyond the normal default settings. The unsigned byte expression must be a number between 0 and 63 and is used as a mask to filter the SANE exceptions. The SANE halt vector is always enabled, and all halts are received by IIGS BASIC. The mask is used to determine if any specific exception will generate a BASIC error message or be ignored.

ON EXCEPTION and OFF EXCEPTION can be used to handle SANE exceptions in your program. (See "Using ON EXCEPTION" in Chapter 7 and Appendix K.)

EXEC

```
::= EXEC filepath[,OFF][,"argument1{[,argumentn]}]

::= EXEC filepath[,OFF][,sexpr]

::= EXEC #filenum[,OFF]

::= EXEC #filenum[,OFF]
```

The EXEC command causes BASIC to take its input from a sequential text file rather than the .CONSOLE device, normally the keyboard. This sequential text file may contain any BASIC statement, lines of a program, or commands that may be entered in immediate mode. The various uses of EXEC are explained in more detail in the "Automatic Execution" section of Chapter 1.

The two forms of EXEC perform the same overall function but with some useful differences. EXEC **filepath** will always return control to the .CONSOLE device when the EOF of the EXEC file is reached, automatically closing the EXEC file. BASIC uses an exclusive file (#30) for EXEC files (#31 is used for CATALOG).

EXEC **#filenum** selects an open file as the new source for the console input stream, but it chains to the previous EXEC file, if an EXEC was already in progress, returning control to that EXEC upon reaching its own EOF. EXEC **#filenum** chains to the .CONSOLE device if an EXEC is not in progress. If the file type of the EXEC file is not TXT or SRC, a file type error will occur.

The OFF option will suppress the normal immediate mode echo behavior for input lines and not display the input text on the .CONSOLE display screen.

The optional arguments (or string expression) allow substitution of values provided on the EXEC command line for placeholders specified within the EXEC file. Arguments are extracted from the comma-separated list of literal characters following the initial quotation mark, or the value of the string expression. If a string expression is used, it may not begin with a quotation mark character. The quoted argument-list option uses all characters following the quote to the end of line as argument data; a closing quotation mark is optional.

Arguments are separated by commas, and two quotation mark in a row are taken as a single quotation mark when processing the quoted argument data into the EXEC argument buffer. The contents of the string expression are copied into the argument buffer as given (including any embedded quotation mark).

When you create an EXEC file, up to nine formal parameters may be encoded into the EXEC file as a special three character sequence. The sequence is *{x}* where the *x* represents any digit from 1 through 9. The formal parameter will be replaced by the argument whose position in the argument list, counting the first argument on the left as passed from the EXEC command as position one, corresponds with *x*. For example, all the characters up to the first comma become argument one and replace all occurrences of the sequence *{1}* in the EXEC text lines before they are passed to the interpreter.

Arguments can only be passed with EXEC **pathname** and cannot be passed with EXEC **#**; that is, there is only one argument buffer and one set of arguments returned by all formal parameter references. However, an EXEC file invoked by EXEC # from within an EXEC file invoked by EXEC **pathname** (with arguments) will reference the arguments in the argument buffer.

Because an EXEC file executes in immediate mode, it can use the commands available there to control the behavior of the EXEC file. Apple IIGS BASIC allows you to use INPUT #30 to read from the current EXEC file invoked by EXEC *pathname*. As a result you can skip forward conditionally within an EXEC file by using immediate mode INPUT # statements within immediate mode IF ... THEN ... ELSE and DO ... WHILE ... UNTIL constructs.

NEW without the optional program memory size parameter may not be executed within an EXEC file; use

```
DEL 1-65279: CLEAR.
```

EXEVENT@

```
::= EXEVENT@(ubexpr)
```

The EXEVENT@ function returns one of 32 external event entry point addresses available within IIGS BASIC. The address returned may be passed to a toolbox function as the address of an event-handling routine. The argument is a number from 32 through 63, and it specifies which external event-handiling address is to be returned.

An external tool set function must JSL to the address returned by EXEVENT@ when the tool set is called by Task Master polling (activated by TASKPOLL ON). The line number of the event-handling routine in your program is defined with the EVENTDEF statement. The index (event number) in the EVENTDEF statement must match the argument to the EXEVENT@ function. The external entry points dispatch as if the Task Master polling mechanism had returned an event code in the range of 32 through 64.

The highest event numbers should be used first in your programs since Task Master may be expanded in the future to use the lower numbers. See Chapter 7, "Advanced Topics."

EXFN and EXFN_

```
::= EXFN[%'@'&'#'$] libname [(lexpr'@var[{,lexpr',@var}])]
::= EXFN[%'@'&'#'$] _libname [(lexpr'@var[{,lexpr',@var}])]

)PRINT EXFNCalcX(2)*32/256
```

EXFN executes an external assembly-language function loaded by an INVOKE statement, and EXFN_ executes external functions defined by a LIBRARY statement, either of which may return a numeric or string result. The library dictionary contains the **libnames** that EXFN may call. The library dictionary is loaded by the LIBRARY statement for toolbox external functions and by the INVOKE statement for invokable module functions. If the **libname** is not found in the library dictionary an

```
?UNDEF'D PROC/FUNCTION ERROR
```

will be displayed.

EXFN searches the invoked module name dictionary, and EXFN_ searches the toolbox LIBRARY dictionary. Any entry point in the toolbox can be called with EXFN, even if it doesn't return a result on the stack given the proper interface definition in the dictionary. The TDF files supplied with IIGS BASIC are defined so that this is possible, and most functions return the error status as the result of calling a procedure as if it were a function.

Standard tool set procedures return error status in the carry and the A register. When a tool set procedure is called with EXFN_, a zero is returned as the function result when the carry is clear, and the contents of the A register are returned as the result when the carry is set.

If you want to pass an integer argument, include an integer variable in the parameter list, but as just a variable, not as an expression.

To pass the address of numeric variables, use the VARPTR( ) function. String variables are converted to counted strings, and the address is passed for the argument. EXFN and EXFN_ process arguments in the same manner as the CALL statement. See the CALL description in this chapter.

EXP

```
::= EXP(aexpr)
```

```
)PRINT EXP(3)
20.0855
)
```

EXP raises e (to 6 places, e equals 2.718282) to the power indicated by the argument value.

EXP1

```
::= EXP1(aexpr)
```

```
)PRINT EXP(3)
19.0855
)
```

EXP1(**x**) accurately computes $e^{\mathbf{x}}$ -1. If the argument **x** is small, such as an interest rate, then the computation of EXP1(**x**) is more accurate than the straightforward computation of $e^{\mathbf{x}}$-1 by exponentiation and subtraction.

EXP2

```
::= EXP2(aexpr)
```

```
)PRINT EXP2(3)
8
)
```

EXP2 raises 2 to the power indicated by the argument value.

FILE

```
::= FILE(sexpr[,FILTYP= TXT'SRC'BDF'ubexpr)
```

The FILE function returns the value 1 if the file given by the pathname string expression exists, or the value 0 if the file does not exist. If any error other than file not found is encountered, that error will be displayed. If the optional file type is not specified, then true will be returned for any file type. If the file type is specified, and the file has some other file type, a file type error will occur. The reserved variable AUXID@ will contain the subtype from the directory entry of the file. The function FILTYP(0) will return the file type of the file.

FILTYP

```
::= FILTYP(filenum)
```

The FILTYP function returns the file type of an open file from the BASIC FCB. FILTYP(0) is a special case that returns the file type of the last FILE function call.

FIX

```
::= FIX(aexpr)
```

```
)PRINT FIX(3.333),FIX(-3.333)
3  ,-3
)
```

FIX returns the integral portion of the value of the argument, truncating the fraction of the absolute value of the argument. FIX differs from INT in that FIX does not return the next lower number for a negative argument. Fix is equivalent to the SGN(x) * INT(ABS(x)).

Fixed-point specification

```
        ::= [**] [$] [+-] digitspec
        ::= [**] [+-] [$] digitspec
        ::= [**] [$] digitspec [+-]
        ::= $$[+-] digitspec
        ::= $$ digitspec [+-]
        ::= [++--] [$] digitspec
```

```
)PRINT USING "+###.###"; 3.14159
+ 3.142
)PRINT USING "+6Z.3Z"; 09999
+009999.000
)PRINT USING "+6&.3&"; 09999
+  9,999.000
)PRINT USING "**+6#.3#"; 09999
+**9999.000
)PRINT USING "**$6#.3#-"; 09999
***$9999,000
)PRINT USING "+$6#.2#"; 09999
+  $9999.00
)PRINT USING "$$+6#.3#"; 09999
$+9999.00
)PRINT USING "++6#.3#"; 09999
+9999.00
)PRINT USING "$--6#.3#"; 09999
$9999.00+
)PRINT USING "$$6#.3#+"; 09999
$  9999.00+
```

The fixed-point specification (fixspec) controls the output format of fixed-point numbers with a PRINT USING or PRINT# USING statement. Fixed-point numbers are any numbers displayed without exponents, including integers, long integers, and real numbers.

A digitspec, composed of combinations of the characters #, &, and Z, is used to define the format of the number being displayed. (See the digitspec description in this chapter.)

If you specify a fixspec with a fractional part and apply it to an integer expression, only zeros will appear to the right of the decimal point, unless you use the SCALE function.

The entire field is filled with exclamation points if the number of digits displayed exceeds the number of digits specified to the left of the decimal point.

The following are the fixspec symbols:

+ reserves a character position for the sign. The sign is printed in all cases.

- reserves a character position for the sign. The sign is printed if negative; otherwise, a space is printed.

$ reserves a character position for a dollar sign.

** means print asterisks instead of spaces in unused character positions.

++ reserves the rightmost unused position(s) for the sign (and following dollar sign, if any).

-- is the same as ++, except that the sign is replaced by a space if it is positive.

$$ reserves the leftmost unused position(s) for a dollar sign (and following numeric sign, if any).

You cannot use $$, ++, or -- if you use Z for the digitspec.

Note that the **, if used, must be the first thing in the fixspec and cannot be used if Z is used for the digitspec because Z leaves no unused digit positions. The dollar sign may come next, or the number sign (+ or -). (The sign of the number can be placed after the last digit.)

FN =

```
::= FNvariablename = expression
```

The FN =, or FN LET, statement is a special case assignment statement used inside a multiline function or procedure definition. The variable can be the name of a local variable, a function or procedure argument, or the name of a function. The variable must be in the current local symbol table, otherwise the message

```
?NOT LOCAL ERROR
```

will be displayed.

Except for this restriction, FN LET is the same as the LET statement. Normally, it is used to assign a function its resulting value, but it is also useful to ensure that an assignment within a procedure or function is local, not global.

Good programming practice would suggest that all local variable assignments be done with FN =, and all global variable assignments be done with LET, thus documenting the programmer's intent and providing an error message if an incorrect variable name is inadvertently used.

FOR ... NEXT

```
::= FOR control variable = aexpr1 TO aexpr2 [STEP aexpr3]
::= NEXT [control variable {,control variable}]
```

```
)15 FOR Index=1 to 500 : PRINT Card+INDEX : NEXT Index
```

FOR and NEXT allow a group of statements to be executed a specified number of times. The first control variable given in the NEXT statement must be the same as the one named in the most recently executed FOR statement; the second control variable given must match the second most recently executed FOR statement, and so on. Incorrectly matched FOR and NEXT statements cause a NEXT without FOR error when a NEXT statement is found and a matching FOR loop is not currently active.

When the FOR statement is executed, it searches forward in the program for a matching NEXT statement, properly accounting for nested FOR statements. This forward search only counts NEXT verbs that begin a *statement* (not a program line), ignoring any NEXT verbs embedded in a THEN or ELSE clause of an IF or ON statement. When the search locates a properly nested NEXT, it is considered a match if the control variable is absent.

If the control variable is present and does not match, the search continues forward until a NEXT with a matching variable name is found. During this latter search, no FOR statements are allowed. If a NEXT with a control variable matching the FOR statement is not found or another FOR statement is encountered, a FOR without NEXT error is will occur.

The control variable may be any integer or real variable, but not an array element or a string variable. If the initial value or the computation of the next value of the control variable generates a result that is out of range for the type of the control variable, an overflow error will occur.

Both the value of the TO limit and the value of the STEP clause must be within the range of the type of control variable used. The FOR statement coerces the value of limit and the step to the type of the control variable and will generate an overflow error if they are out of range. This conversion will also round any nonintegral limit or step value used with an integer control variable, without giving any error message.

FRE

```
::= FRE
```

FRE is a reserved variable that returns the amount of remaining unused data segment memory, measured in bytes. See Appendix D, "Interpreter Data Structures," for information about using memory space more efficiently.

Each time that you access FRE, string data memory is compressed to recover unused string space.

FREMEM

```
::= FREMEM(ubexpr)
```

FREMEM is a function that returns other information about available memory, as follows:

FREMEM(0) returns the free memory in the data segment, without first performing the garbage collection to recover unused string space.

FREMEM(1) returns the size of the data segment after performing grabage collection to recover unused string space.

FREMEM(2) returns the amount of memory currently allocated for arrays.

FREMEM(3) returns the amount of memory currently allocated for simple variables (not including any local variables).

FREMEM(4) returns the size of the current program.

FREMEM(5) returns the size of the program memory segment.

FREMEM(6) returns the size of the library dictionary segment.

FREMEM(7) returns the Memory Manager's unallocated memory total (and does a CompactMem without unlocking any BASIC memory segments).

FREMEM(8) returns the size of the Memory Manager's largest free contiguous block.

FREMEM(9) returns the total memory installed in the system (excluding the 64K dedicated to the sound generator).

GET#

```
::= GET# filenum[,[length] [,recnum]]; stvar
```

The GET# statement reads a record from the random-access file given by **filenum** from the record given by **recnum** and stores the data in the structure array beginning at the specified element. The last parameter can only be a structure variable reference (with a subscript).

The **recsize** of the file, set when the file is opened, determines the read size, unless you use the optional length parameter. The read size is limited by the leftmost dimension of the structure array and the beginning element given by **stvar.**

GET$

```
::= GET$ [#filenum [,recnum]] ; svar
```

```
)110 GET$ Press$
```

GET$ is used to assign a single character or numeral from the keyboard to a string variable in your program, without displaying it on the screen and without requiring that the Return key be pressed.

GET$ with the **#filenum** option will read a single byte from the specified file and assign it to the string variable. Since files can contain values that are not defined as ASCII characters, it is your responsibility to ensure that the file contains valid characters. Getting a byte with a value of zero and embedding it in a string will cause unpredictable results later when using that string.

GET$ treats Control-C like any other character; it does not interrupt program execution. GET$ cannot be used with a numeric variable.

If the program that uses GET$ was called by an EXEC file, the input will be taken from the EXEC file instead of from the keyboard.

GSB.HELLO

GSB.HELLO is the name of the startup program for Apple IIGS BASIC. The prototype interpreter attempts to find the file GSB.HELLO during initialization. If it finds the file (using prefix 0), the file is assumed to be a IIGS BASIC program file, and an implied

```
RUN GSB.HELLO
```

is executed. If the file with the name GSB.HELLO is a IIGS BASIC program, it will commence execution, assuming no errors occur during loading. If a file with this name is not found, the interpreter enters immediate mode after displaying the copyright notice.

GOSUB

```
::= GOSUB linenum'label
```

```
)287 GOSUB 1158
)287 GOSUB READRECORD
```

GOSUB causes program execution to branch to the line indicated by the **linenum** or **label**. When a RETURN statement is encountered, execution branches to the first line following the most recently executed GOSUB statement.

Nesting subroutines more than 40 deep causes a stack overflow error.

GOTO

```
::= GOTO linenum'label
```

```
)GOTO 65200
)100 GOTO STRIKES
```

GOTO causes program execution to branch to the line indicated by the **linenum** or **label**. You can also use it in immediate mode to begin executing a program presently in memory at a given point.

HEX$

```
::= HEX$(aexpr)
```

```
)PRINT HEX$(780)
)PRINT HEX$(-1024)
```

HEX$ returns an eight-character string that is the hexadecimal (base 16) equivalent of the value of the given arithmetic expression. The expression must be in the range +/- 2^32-1; otherwise, an illegal quantity error will result. Eight digits are always returned, filled with leading zeros as necessary.

GRAF

```
::= GRAF INIT iexpr
::= GRAF OFF
::= GRAF ON
```

GRAF ON and GRAF OFF issue QuickDraw II tool set function calls of the same name, thereby switching the Super Hi-Res screen on and off, respectively. GRAF INIT must be executed before using GRAF ON or GRAF OFF; otherwise, a tool set call error will occur.

GRAF INIT must be used before making any QuickDraw II calls through CALL or CALL% to allocate zero-page memory and initialize the QuickDraw mode. The mode is given by **iexpr,** and it must be 0, 320, or 640. GRAF INIT does not execute a LOAD1TOOL call to ensure a minimum revision level, nor does it load the QuickDraw Auxiliary tool set. GRAF INIT immediately issues a GRAF OFF call after the startup call. You must issue a GRAF ON call to activate video display of the graphics screen.

If GRAF INIT has already been issued, a subsequent GRAF INIT will properly stop QuickDraw II and reactivate it with the new mode. INIT mode 0 indicates that QuickDraw II should not be reactivated. GRAF INIT also checks to see if the Window Manager, Menu Manager, and Control Manager are activated. If they are, it issues the proper calls to inform each tool set of the mode change.

HLIST

```
::= HLIST [linenum][ , ' - [linenum2] ]
```

HLIST is a variation of the LIST command that executes a HOME:LIST command sequence. HLIST can only be used in immediate mode. (See the LIST description in this chapter.)

HOME

```
::= HOME
```

HOME clears all text within the current text window and moves the cursor to the upper-left corner of the window.

HPOS and VPOS

```
::= HPOS ' VPOS
```

The HPOS and VPOS modifiable reserved variables contain the vertical and horizontal positions, respectively, of the current print position. Changing their values will change the current print position (and the cursor's position). The position of the cursor can be found by accessing the values of VPOS and HPOS.

Assigning values greater than the height of the text window to VPOS causes the cursor to move to the bottom screen line within the window. Assigning values greater than the width of the text window to HPOS causes the cursor to move to the right margin of the window. The value 0 is converted to the value 1. Assigning values outside the range of 0 to 255 to either VPOS or HPOS causes an illegal quantity error.

IF ... THEN and IF ... GOTO

```
::= IF lexpr GOTO linenum'label[:ELSE linenum'label'statementlist]
::= IF lexpr THEN linenum'label'statementlist[:else statement
or
::= IF lexpr THEN linenum'label'statementlist
 ELSE linenum'label'statementlist                    (must begin next line)
or
::= IF lexpr
  THEN linenum'label'statementlist                   (must begin next line)
    ELSE linenum'label'statementlist                 (must begin next line)
```

```
)IF A=4 GOTO 473
)IF KP+BH GOTO 3785
)100 IF G& MOD F& >2 GOTO 121
)IF 0 THEN PRINT 1
)50 IF 2+2 THEN PRINT.RPT
)IF S/3>=17 * NOT 2 THEN GOSUB 3000 : INVERSE : PRINT "Hi"
)IF X=1 THEN Y=2 : ELSE Y=3
)IF 3<PL5 THEN PL5=-PL5 : ELSE NORMAL : GOTO 376
)718 IF NOT Y THEN 3200 : ELSE TEXTPORT 1,1 TO 4,4 : GOTO 457
```

If the expression following IF evaluates to nonzero (true), the instructions following THEN or GOTO in the same line will be executed. If the expression evaluates to zero (false) execution will continue with the next line.

A string variable or expression is also allowed as the logical expression of an IF statement. The expression is considered true if the string length is nonzero and false if the string is a null string; it is treated as if you had entered LEN(**sexpr**).

In an IF ... THEN statement, the instructions can be a line number or label to which execution should branch or a statement list for BASIC to execute.

In an IF ... GOTO statement, the instruction must be a line number or label to which execution should branch. If the **linenum** or **label** does not exist, an undefined statement error will occur.

The optional ELSE clause in IF ... THEN statements allows you to specify instructions for BASIC to execute if the truth value of the logical expression is false. In other words, when the expression is false, instead of having execution pass to the next higher numbered line, you can have BASIC execute some instructions. The instructions following the reserved word ELSE can be a line number or label to which execution should branch or a statement list to execute. If the logical expression is true, the ELSE clause and any statements following on that line are ignored.

The IF ... THEN ... ELSE statement can be continued on multiple lines as long as each program *line* begins with either THEN or ELSE. Multiline IF ... THEN ... ELSE statements can be nested, and embedded IF statements can also be continued on multiple lines according to the above rule. You must be careful to always include the ELSE clause for any nested IF statement (even if it does nothing) if you want to pair an ELSE line with an earlier IF.

When you break an IF statement into multiple lines at the ELSE verb, the THEN clause cannot be continued any further. Any single IF ... THEN ... ELSE construct can be three lines at most.

Note that if you compare a numeric expression to a numeric variable using a conditional statement, you might not get the results you expected. IIGS BASIC is implemented using the SANE math engine, which does floating-point binary arithmetic with the equivalent of 19 digits of decimal precision (extended precision). Apple IIGS BASIC calculates expressions to 19 digits. However, single- and double-precision variables only have 7 and 15 digits of precision, respectively.

When you compare an expression to a variable, the more precise expression result will not compare as EQUAL or NOT EQUAL to a variable into which that same expression were stored. This can be explained by the following example:

```
10 D#=1/3
20 IF D#=1/3 THEN PRINT "True" : ELSE PRINT "False"
```

You might expect this program to print True but it does not because

```
D# = .3333333333333330000
```

while

```
1/3 = .3333333333333333333
```

and these are in fact two different numbers.

Further details of the issue are explained in Capter 4 and Appendix K.

INDENT

INDENT is a reserved variable that contains the number of spaces to be used to indent FOR ... NEXT loops in the program listings. Its default value is 2.

INIT

```
::= INIT diskname,volumename
```

INIT is used to initialize a disk volume in the disk drive designated by the **diskname** with the given **volumename.** In immediate mode, the INIT statement checks to see if a volume, already present in the indicated drive, is about to be erased. If an existing volume is present, the event queue (typeahead buffer) is flushed and the message

```
Press Y if you want to destory /volname ?
```

is displayed. You must type a Y (or y) from the keyboard before the initialization will begin. If any other character key is pressed the command will be aborted. In deferred mode, the INIT command does not check for a volume before initialization (formatting) begins; it is assumed that the program will check for itself and display appropriate warnings to the user.

INPUT

```
::= INPUT [string ,';] var{,var}
```

```
)1000 INPUT INPUT Zoo$,Gnus,Tolls
)20 INPUT "Enter your age in years"; AGE
```

INPUT accepts numbers or text typed at the keyboard and assigns their values to variables specified in the INPUT statement. INPUT can be used in deferred execution only.

You may optionally include a string in an INPUT statement. The optional string must be a sequence of characters in quotation marks, followed by a comma or semicolon; it cannot be a string variable or expression. When the optional string is present, it is displayed exactly as specified; no question mark, spaces, or other punctuation are displayed after the string. You can use only one optional string.

You can halt program execution during an INPUT statement by pressing Control-C any time during your response (unless BREAK is OFF). You need not press Return or Control-C as the first chararcter of your response; the Control-C is recognized immediately whenever it is typed. If only the Return key is pressed when a string response is expected, the response is interpreted as a null string.

Two special cases exist for entry of numeric representations of mathematical concepts supported by SANE. The characters INF are used to enter either + or - infinity. The characters NaN are used to enter a not a number. The characters NaN can be followed by a constant of one to three digits enclosed in parentheses. The various values determine the type of the NaN that is created, as described in Appendix K, "SANE Considerations."

INPUT#

```
::= INPUT# filenum [,recnum] [;var[{,var}] ]
```

```
)INPUT# 2; Payment%, Grease$
)INPUT# 8, 34; DG(0), DG(2), DG(4)
```

INPUT# reads a line of text from the input file and inputs variables from the text in order, from left to right. The line of text may or may not be terminated by an ASCII carriage return. INPUT # will read the next 255 characters from the file if it does not find a line ending with a carriage return. The text must contain valid constants for the type of each variable, separated by commas. If the initial input line does not contain enough constants for all the variables, the second and subsequent lines will be read from the file until all variables are assigned a value or an EOF occurs.

Automatic conversions are performed for numeric variables, but string constants must be used for string variables. The file to be read from is defined by a file reference number following the reserved word INPUT#. If the file reference number is followed by a comma, the arithmetic expression following the comma specifies a record number at which to begin file access.

INPUT USING

::= INPUT USING **linnum ; svar**

INPUT USING executes the User Input Routine, using the parameters in an IMAGE statement. The User Input Routine is the same routine used by the EDIT command and for entering command lines in GS BASIC. The behavior of the input routine can be customized for your programs with the IMAGE statement parameters. **Linnum1** or **label1** points to the IMAGE statement.

The IMAGE statement for INPUT USING is similar to the one used for PRINT USING, but instead of specs for each variable, it contains a fixed-format sequence of setup parameters. The string variable, **svar,** is both input to and output from the INPUT USING statement. The value of the string is the default value of the line to be edited by the user; it may be a null string if new data are being entered. The edited characters of the line (if any) are returned in place of the default value.

You can control many aspects of the entry process with the parameters in the IMAGE statement, as explained in the section titled "The INPUT USING Statement" in Chapter 7. The IMAGE statement parameters are summarized here for quick reference.

INPUT USING 0; **svar** will execute INPUT USING with the parameters from the prior INPUT USING with a new default data string.

IMAGE **maxlen**,**x**,**y**, **scrnwidth**, **fillchar**, **cursor-mode**, **short**, **long**, **control**, **immediate**, **beep**, **bord_ch**, **0**,**n**, $\mathbf{tchar_1}$,$\mathbf{modfr_1}$,$\mathbf{tmode_1}$ {,$\mathbf{tchar_n}$,$\mathbf{modfr_n}$,$\mathbf{tmode_n}$}

The UIR( ) function provides the status information from the User Input Routine after INPUT USING completes. The UIR function 0, exit type, is the index of the **tchar,** or termination keypress, that ended the input editing. It will be in the range 1 through n, and indicates which **tchar** in the IMAGE statement that was entered.

These are some of the UIR function results: UIR(0) returns exit type, UIR(1) returns last keypress, and UIR(2) returns modifier of last keypress.

INSTR

```
::= INSTR(sexpr, sexpr [,aexpr])
```

```
)PRINT INSTR("Rain in Spain on the Plain", "ai")
2
)PRINT INSTR("Rain in Spain on the Plain"; "ai", 5)
11
```

INSTR searches for occurrences of a specified substring within a string and returns the number of the first character of the substring.

The optional arithmetic expression specifies the character position where the search should begin. If no arithmetic expression is specified, the search begins with the first character of the string expression. If the search fails, 0 is returned.

If the arithmetic expression is greater than the length of the string expression or less than 1, then an illegal quantity error occurs.

INT

```
::= INT(aexpr)
```

```
)PRINT INT(3.3)
3
X=INT(-3.3) : PRINT X
-4
)
```

INT returns the largest whole number value less than or equal to the argument value. We use **whole number** rather thatn **integer** in this definition because the INT function actually returns a real number.

Inteeger constants

```
::= [+`-]{digit}
```

An integer constant is any positive or negative whole number without a decimal point. IIGS BASIC has single, double, and long integers. IIGS BASIC converts any integer constant with nine or fewer digits from characters into binary when a program line is entered. Integer constants with ten or more digits are left as characters in the program and are converted to binary each time the statement is executed. In addition, integer constants in a DATA statement are left as characters and converted to binary by the READ statement. The tokenized, or binary form, of an integer constant is usually smaller than its representation as characters and is never larger.

Your program will execute faster if you use integer constants with nine or fewer digits. Place large integer constants in a DATA statement and convert them into a long integer variable once, with a READ statement.

The expression evaluator has both integer and floating-point mathematical routines and will use the smallest integer size that will represent the value of an expression. Integers values may be freely mixed with real variables and constants. A single integer can represent values in the range of -32768 through 32767, a double integer can represent values in the range of -2147483648 through 214783647, a long integer can represent values in the range of -9223372036854775807 through 9223372036854775807. Attempting to assign a value beyond the respective range to each type of integer variable generates an overflow error.

INVERSE

```
::= INVERSE
```

INVERSE sets all subsequent display to black letters on a white background. Characters on the screen before the execution of the INVERSE or NORMAL statement are not affected. When the video display is a color monitor, the terms **black** and **white** in this description become the **background** and **foreground**, respectively.

INVERSE has no effect on characters read from or written to files.

INVOKE

```
::= INVOKE
::= INVOKE pathname [{,pathname}]
::= INVOKE APPEND pathname [{,pathname}]
```

```
)INVOKE FP1, FP2, /Floppy2/Subr/FP3
)10100 INVOKE Fastprint
```

INVOKE loads an OMF file, as defined by the Apple IIGS Object module format specification. The files given in an INVOKE statement are located and loaded in sequence. If any file is not found or is the wrong type, a file not found error or file type error will occur.

Invokable module files must be created as described in Appendix I. They must contain a static code segment as segment #1 and a private data segment with the segment name DICTIONARY. The dictionary segment defines the interface definitions required by the PERFORM and EXFN statements.

INVOKE accumulates all the dictionary segment entries into a single entry point definition table, maintained in the library memory segment. The entries are hashed into 32 search threads to minimize search time during PERFORM and EXFN.

It is suggested that only one file be loaded per INVOKE statement to allow error handling and use of the RESUME statement. INVOKE will not request volume mounting if the pathname refers to an off-line volume.

Executing INVOKE without any pathname erases all external subroutines previously loaded by other INVOKE statements and releases their memory back to the Memory Manager.

Executing INVOKE without the APPEND option first erases all existing invoke segments and releases their memory back to the Memory Manager before loading the new file or files.

INVOKE APPEND loads the file or files without releasing any previously loaded modules.

Warning:

INVOKE APPEND does *not* prevent duplicate loading of the same module. You must be careful when testing a program not to repeatedly execute INVOKE APPEND statements for the same file.

One approach to handling this problem during program testing is to invoke your modules from immediate mode and only insert the INVOKE statements into your program when you have finished testing. This can easily be done by using an EXEC file to load your program and the invokable modules.

Another approach might be to attempt to use a dummy procedure or function with the EXFN or PERFORM statement without first executing the INVOKE, and execute the INVOKE only after the undefined procedure or function error is trapped with ON ERR.

JOYX and JOYY

```
::= JOYX(ubexpr)
::= JOYY  (a reserved variable)
```

JOYX reads two of the four game paddle inputs (if they are plugged in) specified by **ubexpr**. The unsigned byte expression must result in a number from 0 to 2; otherwise, an illegal quantity error occurs. JOYX returns the value for the paddle given by **ubexpr,** and a reserved variable JOYY is set with the value of the paddle **ubexpr**+1. This function eliminates the interaction between paddles caused by the coupling of the hardware one-shot timers by timing both paddles in parallel. Both JOYX and JOYY return a result with 8 significant bits.

KBD

```
)ON  (KBD)-64  GOTO  100,200,300
```

KBD contains the ASCII value of the last key struck. (See Appendix A, "ASCII Character Codes.")

When you use the reserved variable KBD in an ON ... GOTO or ON ... GOSUB statement, you must enclose KBD in parentheses, or BASIC will not treat it as a variable.

LEFT$

```
::= LEFT$(sexpr, aexpr)
```

```
)PRINT LEFT$("Appleskin",5)
Apple
)PRINT LEFT$("Sparkling",3)
Spa
```

LEFT$ returns a string of specified length composed of the leftmost characters of the given string expression.

If the value of the arithmetic expression exceeds the length of the string expression value, all the characters of the string expression value are returned. If the string expression value contains more than 255 characters, a string too long error results. The value of the arithmetic expression is rounded down to the nearest whole number if necessary. It must be in the range of 1 to 255, or an illegal quantity error results.

LEN

```
::= LEN(sexpr)
```

```
)PRINT LEN("ABCD")
```

```
)PRINT LEN(Yarn$)
```

LEN returns an integer value equal to the length of the string expression, in the range of 0 to 255. A string expression containing more than 255 characters causes a STRING TOO LONG ERROR.

LET

```
::= [LET] var'modifiable  resvar = \expression\
```

```
)LET Henry=FatherofJack
)LET WaterAnimal$="Blue whale"
```

The variable name to the left of the equal sign is assigned the value of the expression to the right of the equal sign. Only one assignment may occur per statement. LET is optional.

- *NOTE:* The LET expression is the only context in which a multiline function, defined with DEF FN ... END FN statements, can be referenced. Referencing a multiline function in any other statement will cause a multiline function reference error.

LIBFIND

```
::= LIBFIND svar,sivar1,sivar2,sivar3
```

LIBFIND searches the library dictionary, loaded with the LIBRARY statement. If the string variable, **svar**, is not a null string, the library is searched for the **libname** given by the value of the string. The tool set function number for the **libname** is returned in **sivar1**, the tool set tool number is returned in **sivar2,** and the result stack size is returned in **sivar3**. If the **libname** is not found, all three integer variables are set to zero.

LIBRARY

```
::= LIBRARY [filepath[{,filepath}]]
```

The LIBRARY statement loads one or more tool set Definition Files into the library dictionary. The library dictionary is a separately allocated memory segment that contains a dictionary of interface definitions for one or more tool sets. Each interface definition contains the function or **libname** name, tool number, function number, parameter count and type for each parameter, result word count, function result type, and error mode.

If there is not enough free memory for all the TDF dictionary data, an out of memory error will occur. Entries are also inserted into the dictionary by the INVOKE statement. If no filenames are present, the library dictionary is deleted, except for entries inserted by the INVOKE statement. The TDF for most tools are supplied with IIGS BASIC. More information about the format of a TDF may be found in Appendix H.

LIST

```
::= LIST [linenum1'label1] [ ,'- [linenum2'label2] ]
```

```
)LIST
)LIST 5 - 300
)LIST 5,300
)LIST 5-300
)LIST start,finish
)LIST start-2000
)LIST - 2100
)LIST 1585 -
```

LIST displays the contents of the program currently in memory.

The first example above displays the entire program currently in memory. The next three examples display lines 5 through 300, (assuming that they exist), of the program currently in memory. The next two examples show how you may use line labels instead of line numbers. The last two examples will list, respectively, from the beginning of the program to line 2100 and from line 1585 to the end.

You can stop the listing by pressing the space bar and restart it by pressing any other character key. Pressing Control-C terminates the listing.

LISTTAB

```
::= LISTTAB
```

LISTTAB is a modifiable reserved variable that is set each time a program is loaded. Each IIGS BASIC program contains a header that is not part of the program statements, and LISTTAB is set from this header information. LISTTAB causes all the lines of a program to be indented a fixed amount when it is listed by the LIST command. LISTTAB allows you to to offset the program statements to the right of line labels and make your program easier to read.

The default value of LISTTAB is 5, which effectively disables label indenting because the line numbers are printed right-justified and require five characters plus a space. You should set LISTTAB to the length of your longest label plus 6 to align the left margin of all the statements in a listing.

The LISTTAB modifiable reserved variable also controls a special LIST statement display mode. When the value of LISTTAB is larger than 127, the line numbers are not output in the listing and left margin setting is taken as the value LISTTAB-128. Thus, a listing of a IIGS BASIC program without line numbers can be diplayed, printed, or output to a text file.

LOAD

```
::= LOAD pathname
```

```
)LOAD Countdown
)LOAD 2/Somefile
```

LOAD reads a specified BASIC program from a disk file and stores it in memory. The pathname of the program to be loaded must follow the reserved word LOAD. (See the Chapter 5, "File Handling" for an explanation of pathnames.)

All variables in the loaded program are cleared; numeric variables are all set to zero, and string variables are set to null strings. All files are closed, with the exception of any EXEC file being executed. Any existing program is cleared from memory.

Attempting to load a file other than a BASIC program causes a file type error.

IIGS BASIC stores the program in a separate memory segment, and LOAD attempts to allocate a segment large enough for the program plus a small amount of extra space for additions. The program segment is automatically extended as new lines are entered. BASIC attempts to extend the program segment using unallocated memory before it begins shrinking the user data segment.

BASIC does not shrink the program segment once a large program has been loaded, chained, or run, even if the current program is smaller that the program segment size. The only way to recover the unused space in the program segment is to use the NEW statement with the program segment size option.

LOCAL

```
::= LOCAL varname{,varname}
```

The LOCAL statement can only be used within a function or procedure definition. If LOCAL is used elsewhere in a program, executing it will generate `not local error`. LOCAL adds additional local variables to the current local variable table for use during the execution of a multi-line function or procedure. All local variabls are transient and do not retain their values from one execution of a function or procedure to the next. Local arrays are not allowed.

A local variable table is created each time a function or procedure is executed by a FN**name** (...) or PROC (...) reference. The arguments of the function or procedure are always inserted into the local variable table as local variables. In addition, a local variable with the same name as the function is created to receive the resulting function value by an FN= assignment.

Only the most recently created local variable table is accessible during program execution. In other words, a function or procedure can only refer to its own local variables and the global variables, but not the local variables of any other procedure or function, even if it was itself executed by another procedure or function. The local variable tables are allocated using a stack and require free memory in the data segment.

When BASIC is searching for a variable in the variable tables, the local variable table is always searched before the global variable table. If a variable name is used in both the current local variable table and the global variable table, the local variable will always be found and used in place of the global variable.

A procedure or function definition may contain one or more LOCAL statements, and they may occur anywhere within the definition. Normally, LOCAL statements should immediately follow the DEF statement, but if they do not or are embedded in conditional (one-shot) logic a given variable will be global until it is defined in a LOCAL statement, and thereafter it will be a local variable even if the procedure or function statements loop back to a statement where the variable was initially a global reference. In other words, BASIC does not look ahead at function or procedure definitions for all the LOCAL statements.

❖ *Note:* If you create programs that depend upon this interpreter behavior, your programs will not compile or function correctly if a IIGS BASIC compiler becomes available (compilers usually require that all the LOCAL definitions precede all other statements).

LOCATE

```
::= LOCATE [row][,column]
```

LOCATE positions the text console cursor to the row and column specified by the arguments. If only the row is given the column remains unchanged, and given, the row is unchanged. The row argument must be within the range 1 through 24, the column argument must be in the range of 1 through 80. The value 0 is treated as the value 1.

The location specified by LOCATE is relative to the upper-left corner of the current textport. The LOCATE statement duplicates the function of assignments to the HPOS and VPOS modifiable reserved variables.

LOCK and UNLOCK

```
::= LOCK pathname
::= UNLOCK pathname
```

```
)LOCK Barndoor
)UNLOCK Secrets
```

LOCK prohibits writing to, saving, or deleting the file named as its argument. Locked files are shown with an asterisk to the left of their file type when cataloged. Volume names can not be locked, but subdirectories can be.

UNLOCK allows you to remove the protection from a locked file that you want to delete, rename, change, or save. The reserved word UNLOCK must be followed by the file's pathname.

LOG

```
::= LOG(aexpr)
```

```
)PRINT LOG(20.0855)
3
)
```

LOG returns the natural (base e) logarithm of the argument value.

LOGB%

```
::= LOGB%(aexpr)
```

LOGB% returns the binary exponent of the argument value as a signed integer.

LOG1

```
::= LOG1(aexpr)
```

```
)PRINT LOG1(20.0855)
3
)
```

LOG1 returns the natural (base e) logarithm of one plus argument value. LOG1(**x**) accurately computes LOG1(1+**x**). If the input argument **x** is small, then the computation of LOG1(1+**x**) is more accurate than the straightforward computation of LOG1(1 + **x**) by adding **x** to 1 and taking the natural logarithm of the result.

LOG2

```
::= LOG2(aexpr)
```

LOG2 returns the base 2 logarithm of the argument value.

Logical expressions

Logical expressions are also called **relational expressions** and **Boolean expressions**. They are similar to arithmetic expressions, but use different operators. A logical expression has a value of either 1 for true or 0 for false. Any arithmetic expression with a nonzero value has a truth value of true, and any with a value equal to zero has a truth value of false. When IIGS BASIC returns a true or false result for a logical expression, it creates an integer 0 or 1.

The following are the eleven logical operators:

Symbol	Meaning	Example	Truth value
=	Equal to	3=3	1
<	Less than	3<1	0
>	Greater than	7>4	1
<= or =<	Less than or equal to	5<=4	0
>= or =>	Greater than or equal to	8>=5	1
<> or ><	Not equal to	4<>4	0
<=> or >=<	Ordered (vs unordered)	A<=>NaN	0
AND	Conjunction	5 AND 0	0
OR	Disjunction	8 OR 3	1
XOR	Exclusive OR	8 XOR 3	0
NOT	Negation	NOT 4	0

You can use all the logical operators, except ordered (<=> or >=<), in string expressions. For example, "alpha" < "beta" is true.

The ordered operator is used to test for the SANE NaN. It will return false (0) if either operand is a NaN and true (1) if both operands are valid numbers. NaNs are created by attempting various mathematically meaningless operations, such as dividing zero by zero, adding -infinity to +infinity, or trying to take the square root of a negative number.

Long integers

```
::= [+'-]{digit}
```

Long-integer constants may be up to 19 digits long. You can mix long-integer constants and variables in arithmetic expressions with single or double integers or reals (See the description of reals in this chapter). Long-integer variable names must end with an ampersand (&).

A long integer can represent values in the range from -9223372036854775807 to 9223372036854775807. Exceeding this range causes an overflow error. The binary value that would represent the number -9223372036854775808 is used to represent the SANE NaN that results when various mathematical operations generate an unrepresentable number. (See Appendix K, "SANE Considerations" for more details about NaN.)

.MEMBUFR

.MEMBUFR (pronounced "dot-mem-buffer") is a special pseudo-character device that provides a 255-byte memory buffer that you can use for high-speed I/O without having to create a disk file. .MEMBUFR can be used in an OPEN statement to associate it with a file reference number and then used with PRINT# or PRINT USING # and then INPUT # to create exactly formatted strings.

There is only one actual memory buffer, and it has only one current position pointer. The current position pointer is set to zero when a carriage return is sent to the device and advanced one with each character output to or input from the device. If more than 256 characters are sent to .MEMBUFR the current position pointer wraps around to zero and will overwrite the beginning of the buffer.

Because the memory buffer used by .MEMBUFR is shared with the BASIC EDIT, INPUT USING, CAT, CATALOG, DIR, and TYPE statements, you must print to it and input from it without using any of these commands in between.

MENUDEF

```
::= MENUDEF index,linnum'label [{,linenum'label}]
```

MENUDEF defines for IIGS BASIC the menu-item-handling routines to use when Task Master returns a menu-select event. The menu item identification numbers defined by the Menu Manager must be assigned values from 256 through 383. IIGS BASIC uses the menu item identification number minus 256 as the index of its internal menu-item dispatch table.

MENUDEF is used to define the entries in the internal table. The **index** parameter is a number from 0 through 127 that selects which entry to define (or reset with a 0). If multiple line numbers are used, each one defines the line number of the next entry in the menu -item dispatch table. You need not define consecutive identification numbers, but they must fall within the above range.

MENUDEF is only relevant when you are also using the TASKPOLL statement in conjunction with the Window and Menu Managers.

MID$

```
::= MID$ (sexpr, ubexpr1 [, ubexpr2])
```

```
)PRINT MID$("Bookkeeping",5)
)keeping
)PRINT MID$("Bookkeeping",5,4)
)keep
```

MID$ returns a substring of a given string expression. The first unsigned byte expression specifies the first character to be returned from the string, and the optional second unsigned byte expression specifies the length of the substring to be returned.

If the value of the first expression exceeds the length of the string expression value, then a null string is returned. If the value of the second expression is greater than the number of characters to be retrieved from the string expression value, all the characters from the position specified by the value of the first unsigned byte expression to the end of the value of the string expression are returned.

If the string expression value contains more than 255 characters, a string too long error occurs. If the value of either arithmetic expression is outside the range of 1 through 255, an illegal quantity error occurs.

NEGATE

```
::= NEGATE(aexpr)
```

Negate returns the value -**aexpr**. This seemingly simple function is included because of the specialized SANE data type representations for infinity and NaN results (see Appendix K, "SANE considerations"). You should use NEGATE rather than -1***aexpr** to properly negate such a result.

NEW

```
::= NEW
::= NEW iexpr
```

NEW without the size option erases the current program and all its associated variables from the computer's memory and closes all open files, except a text file being executed. (See the description of EXEC in this chapter.)

NEW may be used with an optional integer expression to request that the program memory segment size be changed to the indicated size (rounded up to the nearest even multiple of 256 bytes). When you use the size option, the current variables, and program, file buffers, etc all remain unchanged.

If the current program is larger than the indicated size, the program segment is reduced to the size required to contain the program (rounded up). If unallocated memory is not available to expand the program segment, BASIC will reduce the data segment (if it has enough space.) An out of memory error will occur if the requested memory is not available.

NORMAL

```
::= NORMAL
```

NORMAL is the default display mode. It sets the display to white letters on a black background. Characters on the screen before the execution of the NORMAL statement are not affected. If the video display is a color monitor **white** refers to the foreground color, and **black** refers to the background color, as set with the Control Panel.

NORMAL has no effect on characters read from or written to files.

NOTRACE

```
::= NOTRACE
```

NOTRACE cancels TRACE, stopping the display of the line numbers of executing program statements. There are no options associated with it.

OFF EOF#

```
::= OFF EOF# filenum
```

The OFF EOF# statement cancels an ON EOF# statement. After an OFF EOF# statement has been executed, BASIC resumes displaying error messages and halting execution when an end of file is reached, just as it did before the ON EOF# statement was executed. You must follow the reserved word EOF# with a file reference number to specify which file's ON EOF# statement should be canceled.

ON EOF#

```
::= ON EOF# filenum  statementlist
```

ON EOF# is used to force BASIC to allow your program to control what happens if BASIC reads past the end of a file, just as an ON ERR statement allows your program to perform its own error handling.

A statement or statement list must follow the reserved word EOF#, as in a GOTO statement.

ON BREAK and OFF BREAK

```
::= ON BREAK statementlist
::= OFF BREAK
```

ON BREAK is used to force BASIC to allow your program to control what happens if the Attention character, a Control-C, is entered during input or typed while your program is running. ON BREAK is a special case of the ON ERR statement that allows your program to perform its own error handling.

ON BREAK is provided as a separate statement to simplify the programming of ON ERR for the more important types of errors and to allow a single handling routine for the user Attention function. ON BREAK is also useful for handling a user request to abort a long operation; for example, to stop printing a report when a paper jam occurs.

A statement or statement list must follow the reserved word BREAK, just as with the ON ERR statement. IF the BREAK OFF statement has been executed, typing the Attention character will be ignored and entered into the typeahead buffer like all other characters.

OFF BREAK cancels the most recently executed ON BREAK statement. There are no parameters or options associated with it. After an OFF BREAK statement has been executed, BASIC resumes displaying the

```
PROGRAM INTERRUPTED IN linnum
```

message and halting execution, just as it did before the ON BREAK statement was executed.

❖ *Note:* The ONBREAK and OFFBREAK statements are only efective in BREAKON mode. (See the description of BREAK earlier in this chapter.)

ON ERR and OFF ERR

```
::= ON ERR statementlist

::= OFF ERR
```

```
10 REM EXAMPLE OF ERROR HANDLING
20 ON ERR GOSUB 1000
30 INPUT "Please type a single number between 1 and 100";X
40 PRINT "The number you typed was ";X
50 END
1000 REM ERROR HANDLING SUBROUTINE
1010 PRINT : PRINT "I'm very sorry, but only a number will do. Please try again."
1020 RETURN
```

ON ERR is used to force BASIC to let your program handle any errors that might occur by branching to an error-handling subroutine in your program.

The ON ERR statement should not be used as a tool for finding errors in programs. (Use the TRACE statement for this instead.)

If a program contains more than one ON ERR statement, the statement list of the most recently executed one will be used.

OFF ERR cancels the most recently executed ON ERR statement. There are no parameters or options associated with it. After an OFF ERR statement has been executed, BASIC resumes displaying error messages and halting execution, just as it did before the ON ERR statement was executed. ON BREAK will remain active after OFF ERR has been executed.

❖ *Note:* The statements that ON ERR causes to be executed must themselves be free of errors, or an endless loop may result. The endless loop can be interrupted by Control-C because Control-C is handled separately by ON BREAK. For a complete list of BASIC errors, see Appendix B, Error Messages.

ON EXCEPTION and OFF EXCEPTION

```
::= ON EXCEPTION statementlist
::= OFF EXCEPTION
```

ON EXCEPTION is a separate version of ON ERR for errors that occur in mathematical computations. Apple IIGS BASIC uses the SANE 65816 implementation of the IEEE Standard 754 for binary floating-point arithmetic for real and long integer operations, and a mixed mode operation with one real or long integer operand. OFF EXCEPTION is the default mode for IIGS BASIC, and so computational results may return an infinity or a SANE NaN result. (See Appendix K, "SANE Considerations".)

After ON EXCEPTION has been executed, the statement list will execute if any of these exceptions occur during mathematical expression evaluation or assignment. The following are exceptions:

- an invalid operation is attempted (such as SQRT(-2))
- overflow
- underflow
- divide by zero
- unordered comparison (such as A<B where B is NaN)
- inexact result

EXCEPTION ON is used to select the subset of these exceptions that you want dispatched to ON EXCEPTION prior to enabling exception trapping with ON EXCEPTION.

The error code and error line for ON EXCEPTION are returned in ERR and ERRLIN, as with the ON ERR statement. (See Chapter 7, "Advanced Topics," for details of the codes and other considerations.) Executing OFF EXCEPTION will restore BASIC to its default mode, in which the divide by zero, not a number, and overflow errors will occur or be trapped by ON ERR.

ON KBD and OFF KBD

```
::= ON KBD statementlist

::= OFF KBD
```

```
10 ON KBD GOTO 100 : REM BASIC branches here when any key is pressed
20 PRINT "."; : REM Print periods while not handling key-strokes
30 GOTO 20
100 PRINT KBD : REM Display the ASCII value of the key last pressed
110 ON KBD GOTO 100 : REM Reenable ON KBD. Must be before return
120 RETURN : REM Program jumps back to the statement following the one during which
    a key was pressed
```

ON KBD is used to cause BASIC to execute a specific statement list immediately after any key is pressed. The statement list to be executed must follow the reserved word KBD.

Note that you must reenable the ON KBD statement immediately before executing the RETURN statement at the end of the statement list.

After an ON KBD statement has been executed, BASIC continues executing the program normally -- but as soon as any key is pressed, execution branches back to the most recently executed ON KBD statement. Then the statement list pointed to by the ON KBD statement is executed.

The branch to the ON KBD statement list is treated as a GOSUB to a subroutine, so the program segment that KBD causes to be executed must end with a RETURN statement. To enable ON KBD to handle more than one keystroke, the last statement in the list should be another ON KBD statement.

❖ *Note:* When ON KBD is in effect, the program cannot be halted by pressing Control-C because that keystroke is treated like any other. However, the ON KBD statement could cause a branch to a STOP or END statement if Control-C is pressed. A RETURN statement placed after the STOP would allow the CONT statement to be used.

After ON KBD is executed and a program returns to immediate mode through a STOP or END statement, ON KBD trapping is suspended. If the program is restarted by using the CONT command, the ON KBD trapping will be reenabled. However, if the program is restarted by GOTO or RUN commands, the ON KBD trapping will not be restarted until another ON KBD statement is executed.

ON ... GOSUB

```
::= ON ubexpr GOSUB linenum `label {[,linenum`label]}
```

```
)1000 ON Corfu GOSUB 1000, GOOFOO, 3000, 4000
```

ON ... GOSUB is identical to the ON ... GOTO statement, except that the line numbers or labels following the reserved word GOSUB must be reference subroutine entry points.

ON ... GOTO

```
::= ON ubexpr GOTO linenum `label {[,linenum`label]}
```

```
)1000 ON X GOTO 100, DOIT, 300, 40
```

ON ... GOTO is used to specify different program branch points, based on the value of an unsigned byte expression. The arithmetic expression must follow the reserved word ON, and the line numbers or labels to which execution branches must follow the reserved word GOTO.

If X=**1**, execution branches to the first line in the list (line 100); if X = 2, execution branches to the second line in the list (line DOIT); if X=3, execution branches to line 300 (the third item in the list); and so on.

The value of the arithmetic expression must be within the range of 0 through 255, or an illegal quantity error occurs. If the value of the arithmetic expression is 0, or greater than the number of line numbers or single integers given in the ON ... GOTO statement, the list of line numbers (or labels) is ignored and execution continues with the next statement in the program.

ONTIMER and OFF TIMER

```
::= ON TIMER (aexpr) statementlist
::= OFF TIMER
```

ON TIMER enables event trapping using the 1-second interrupt capability of the Apple IIGS clock. ON TIMER (**aexpr**) sets a countdown interval of **aexpr** seconds long. The interval is given by the arithmetic expression, and it must be a number in the range of 2 through 86400.

The countdown is complete when the interval counter reaches zero. Then the statement is executed when the current program statement completes execution. The statement list must end with a RETURN statement to return control to the next sequential statement in the program.

The TIMER countdown interval is approximate only and does not guarantee a precise amount of time. The first 1-second interrupt from the clock may occur in a few microseconds or in an entire second after enabling the countdown. In addition, some activities, such as disk I/O operations or AppleTalk communications, have higher priority than the 1-second interrupt, and they may lock out the timer interrupt for more than a second.

All of these factors can delay the processing of the ON TIMER event trapping. You should envision you timer intervals as requests for a delay of not less than **aexpr** seconds minus one, not as a request for an exact time delay in seconds. The ON TIMER statement will have no effect unless the 1-second interrupt is enabled by the TIMER ON statement.

OFF TIMER disables the most recently executed ON TIMER statement.

OPEN

```
::= OPEN openpath, [FILTYP=DIR · TXT · SRC · BDF · ubexpr ]
[FOR INPUT · OUTPUT · APPEND · UPDATE] AS # filenum [, recsize]
::= OPEN #filenum1,FOR {INPUT · OUTPUT · UPDATE}
AS # filenum2 [, recsize]
```

```
)OPEN DOOR, AS #6
)OPEN Window, AS#4,163
)OPEN .CONSOLE, FORINPUT AS#2
)OPEN ptr$, FOR OUTPUT AS#1
)3309 OPEN "BINARY", FILTYP=6 FOR APPEND AS#11,256
)10 OPEN #11 FOR INPUT AS#7
```

OPEN is used to open files for access and must precede any file I/O statements accessing a given file. The arguments following OPEN are the file's pathname or characrter device name, the optional file type, the optional mode, the AS file reference number, and the optional record size.

The FILTYP= parameter may be omitted, in which case the file will be opened if its file type is TXT, SRC, or BDF. The file reference number is used in all subsequent I/O statements to refer to the file while it is open. The file reference number can be any arithmetic expression having a value of 1 through 29.

If the reserved words FOR INPUT are present, the file is opened as a read-only file and may not be written to. If the reserved words FOR OUTPUT are present, the file is opened as a write-only file and may not be read from. If the reserved words FOR UPDATE are present, the file is opened as a read-write file and may be written to and read from. If the FOR mode parameter is omitted, the file is opened FOR UPDATE. The access requested for a file must match the access modes permitted in the directory-entry access parameter. If an access mode is locked and that mode is requested, a file locked error will occur (for either read or write access).

The FOR APPEND option is a variant of FOR OUTPUT, and it is used in sequential access to allow PRINT# or WRITE# statements to append new information to the end of an existing file without disturbing any existing data.

The **recsize** parameter can be given for BDF files. It is used to allocate a record buffer for random access. If the **recsize** parameter is not given for a BDF file, the subtype from the directory entry is used as the record length and buffer size.

Record buffers are not allocated for TXT and SRC files.

The second variation of OPEN allows you to open the same file twice. The file referenced by **filenum1** must currently be an open file. **Filenum2** is the **slave file** and **filenum1** is a **master file**. A given master file may have up to seven slaves linked to it.

Only the master file in a master-slave chain of disk files can be used in WRITE mode, all other disk files in the chain must be opened FOR INPUT. The master file in a chain may be opened for OUTPUT, APPEND, or UPDATE; otherwise, all files in the chain must be opened FOR INPUT. Closing the master file in a chain without first closing the slave files will make the slave file with the lowest file reference number the new master for the chain. The new master will remain in INPUT mode.

Opening the same file twice can be a means of accessing a control record at the beginning of the file or referencing an index structure embedded in a data base file. The main benefit of this capability is that each slave file has its own record buffer, thereby allowing access to multiple records from the same file simultaneously. Future versions of ProDOS 16 may also allow multiple systems to share common files through a file server.

A chain of files must all be disk files or character device files; mixing types is not allowed. A chain of character device files may be created regardless of the file mode, but the results may be unpredictable; for example, trying to read from a printer is likely to hang the system indefinitely.

OPEN window

```
::= OPEN VARPTR(parmlist), [FILTYP=expr] FOR OUTPUT
AS #filenum [,bufsize]
```

OPEN window provides a means of linking a Window Manager window port to IIGS BASIC as an output file. The VARPTR( ) function returns the address of a parameter list. (See Chapter 7, "Advanced Topics," for details.)

When opening a window, the parameter list is an extended version of the parameters used with the Window Manager NewWindow call. See the *Apple IIGS Toolbox Reference* manual for the detailed description of the NewWindow parameters in the parameter list.)

OPEN window may also be used to link a file to a QuickDraw II GrafPort by a GrafPort pointer. This variation is selected from the value of the option word in the parameter list.

You must create the NewWindow parameters yourself in a structure array and pass the address of it to OPEN through the VARPTR function. You must use the VARPTR function to indicate that you want OPEN to open a window or GrafPort as a file.

A window (but not a GrafPort) is closed with the CloseWindow call when the file is closed with CLOSE # (or the CLOSE all variation), and the optional buffer is deallocated and the handle discarded.

After a window file has been opened, you can use the PRINT# or PRINT# USING statements to direct output text to the window at its current cursor position. IIGS BASIC does *not* provide for positioning the cursor as it does for the text display. You must do this yourself using the appropriate QuickDrawII calls before executing PRINT# or PRINT# USING.

Each time a PRINT statement directs output to a window file a Window Manager SelectWindow call will precede the QuickDraw II text drawing calls. IIGS BASIC uses DrawChar, DrawString, and DrawCString for output to a window port or a GrafPort. When GrafPort mode is selected, a QuickDrawII SetPort call precedes the drawing function calls.

You must enable QuickDraw II with the GRAF INIT statement prior to opening a window file; otherwise, the message

```
?TOOLSET ERROR =$0400
```

will be displayed when OPEN window is attempted.

OUTPUT#

```
::= OUTPUT# filenum
```

```
)OUTPUT #5
```

OUTPUT# redirects screen output to a specified file. All PRINT, LIST, TRACE, and CATALOG statement output is sent to the specified file, but keyboard input is echoed and error messages are still sent to the screen. The file used for output is specified by its file reference number (set by an OPEN# statement) following the reserved word OUTPUT#.

System I/O devices such as .CONSOLE and .PRINTER are treated as files and can be opened and used as such.

To resume normal screen output, type

```
)OUTPUT# 0
```

and characters will again be displayed on the screen. A CLOSE or CLOSE# statement also redirects output to the screen.

OUTREC

OUTREC is a reserved variable that contains the maximum length of lines output by the LIST command before wrapping the text on the next line by issuing a carriage return. The value of OUTREC must be greater than the value of INDENT. Setting OUTREC to zero suppresses the wraparound of the text displayed by the LIST command, which is useful when listing a program to a text file.

The following command sequence will make a text file of the current program:

```
)OPEN PROGNAM$+".TXT" AS #9 : OUTREC=0 : OUTPUT #9 : LIST : OUTPUT #0 :
 OUTREC=80 : CLOSE #9
```

The three statements OUTPUT# 9 : LIST : OUTPUT #0 must be executed together as a single command line. This is because LIST cannot be a statement in a program, and you must turn off the switching of the console output to file #9 immediately after the LIST statement is complete.

OUTREC is set from the program header whenever a program is loaded by the LOAD, RUN, or CHAIN statements.

PDL and PDL9

```
::= PDL(0 <= ubexpr <=3)
::= PDL9
```

PDL reads the position of the game control paddle (if it is plugged in) and returns a value in the range of 0 through 255. PDL actually reads the position twice as fast as the original Apple II routines and discards the least significant bit, thus eliminating the uncertainty caused by the variable processor speed of the Apple IIGS. The reserved variable PDL9 will return the 9-bit result calculated by the prior execution of the PDL function.

❖ *NOTE:* Reading any two paddles in quick succession will tend to produce unstable results because of the hardware coupling among all four paddles. Using the JOYX function will eliminate this interaction when reading two paddles or both axes of a joystick.

PEEK

```
::= PEEK(iexpr)
```

PEEK reads a byte from memory at the address given by the integer expression and returns an unsigned integer in the range of 0 through 255. The integer expression must be a positive integer less that 2 of ^{24}. Care should be exercised in using PEEK because improperly reading many I/O devices and control registers can crash the system.

Programmers concerned about writing programs that will run on new versions of the Apple II product family should avoid the use of the PEEK function because it contains a hard-coded address that may not be supported in the future. PEEK is provided for those who want to build nontransportable, locked-in programs.

PERFORM

```
::= PERFORM filepath [(lexpr`var[{,lexpr`,var}])]
```

```
)PERFORM ARRAYADD(VARPTR(A1(0,0)),VARPTR(A2(0,0)))
)PERFORM Errproc(R,13-6,VARPTR(D))
```

PERFORM executes a specified assembly-language procedure previously loaded by an INVOKE statement. If an argument list is present (enclosed in parentheses after the procedure name), each argument is evaluated and passed to the procedure before execution. Numeric arguments are converted to the type specified by the interface definition entry in the library dictionary.

The library dictionary contains the names of the procedures that may be performed. The dictionary entry also contains a description of the number, order, and type of arguments required by the procedure or function. The library dictionary entry is read from the dictionary segment of the invokable load file by INVOKE when the assembly-language module is loaded.

To pass real or integer constants or the values of single variables, include them in the argument list. A string or string expression may not be used for a numeric argument or vice versa; attempting to do so will cause an argument type mismatch error. If the proper number of arguments in not supplied, an argument count error will occur.

To pass addresses of a numeric variable, use the VARPTR( ) function.

If you want your subroutine to operate on a BASIC string in memory, simply using a string variable will pass an address pointing to the string's descriptor in memory. The subroutine should be designed to act on the string from that point on. Alternately, you can pass the address of the string data by using the VARPTR$( ) function.

A third choice is also available if you define the argument as a counted string argument in the interface definition. When a counted string argument is required by the procedure, IIGS BASIC creates a counted string from a BASIC string and passes the address of the count byte as the argument.

See Appendix I, for details on how to write invokable modules.

PFX$

```
::= PFX$(ubexpr <= 7) = sexpr
::= PFX$(ubexpr <= 8)
```

PFX$(0 through 8) is a string function that returns the value currently assigned to that prefix by ProDOS 16. In addition to the eight normal prefixes, IIGS BASIC will return the boot-volume-name as PFX$(8). This pseudo-prefix can be referenced by using an asterisk instead of a digit at the beginning of a pathname.

PI

```
::= PI
```

PI is a real reserved variable name that contains the value of π accurate to 20 decimal digits. When used in an expression, it returns a SANE extended-format representation of π. PI may be converted to single or double precision by assigning it to a variable.

POKE

```
::= POKE iexpr,ubexpr
```

POKE stores a byte, given by the unsigned byte expression, at the address given by the integer expression. The integer expression must be a positive number less than 2^{24}; otherwise, an illegal quantity error results.

Programmers concerned about writing programs that will run on new versions of the Apple II product family should avoid the use of the POKE statement because it contains a hard-coded address that may not be supported in the future. Invokable modules are a preferable alternative to most of the uses associated with the POKE statement.

POP

```
::= POP
```

POP allows you to jump out of one level of subroutine nesting by removing the top pointer from the program stack and discarding it. When the next RETURN statement is encountered after a POP statement is executed, instead of branching to the first statement beyond the most recently executed GOSUB, BASIC branches to the first statement beyond the second most recently executed GOSUB.

PREFIX

```
::= PREFIX
::= PREFIX ?
::= PREFIX pfx
::= PREFIX directory pathname
::= PREFIX pfx,directory pathname
```

The PREFIX statement provides two functions. The first three forms will display the current prefixes on the next lines of the screen, with the first form being equivalent to PREFIX 0, and PRINT PREFIX$. PREFIX ? will display all eight prefixes in order, one line per prefix.

The **pfx** prefix selector is a single digit from 0 through 7. The fourth and fifth forms will set prefix 0 and the **pfx** prefix, respectively.

PREFIX$

PREFIX$ is a modifiable reserved variable that contains the most recently assigned default pathname prefix, known as prefix 0, and PFX$(0).

PRINT

```
::= ?'PRINT {[,';] [expr]} [,';]
```

```
)PRINT
)PRINT "Several words of text."
Several words of text.
)A$="E is about " : E=2.718
)PRINT A$;E
E is about 2.718
```

PRINT displays text. An item list may include any expression, comma, semicolon, TAB specification, or SPC specification following the reserved word PRINT.

BASIC evaluates expressions in PRINT statements and displays their values. If there are several expressions, their values are displayed in sequence. A PRINT statement without an item list moves the cursor to the beginning of the next screen line.

If a comma separates two expressions, a tab action separates their values on the screen. If a semicolon separates them, the second value is displayed after the first, with no intervening spaces.

Following the last expression in a PRINT statement, there may be a semicolon, comma, or nothing. If there is nothing after the last expression, the cursor moves to the beginning of the next screen line. A comma causes a tab action. A semicolon leaves the cursor in the position immediately following the last character displayed.

All numeric values are formatted uniformly, regardless of the variable type. The reserved variable SHOWDIGITS controls how many significant digits will be displayed, and the magnitude of the value generally controls the format. If the value is greater than or equal to $10^{-showdigits}$ and less than $10^{showdigits}$, the expression or variable is formatted in fixed-point format; otherwise, the number is formatted in scientific notation.

PRINT will format any integer variable in scientific notation if SHOWDIGITS is smaller than the number required to print the number as an integer. Likewise, PRINT will format any real variable in fixed-point notation if the value is within the above range *and* if formatting the number in fixed format would not result in a loss of precision for the SHOWDIGITS number of significant digits.

For example, assume that SHOWDIGITS equals 7, and the value is 1E-7, or .0000001. This value is the smallest number that can be formatted in fixed format, but if the value were 1.1E-7, it would be .00000011 in fixed format, with eight digits (even though only two of them are significant digits). Rather than drop the significant digits BASIC formats the numbers in scientific notation.

PRINT USING

```
::= ?'PRINT USING linenum'string'svar [;expr[{,expr}]] [;]
```

The PRINT USING statement is the same as a PRINT statement with a USING clause, used to control the format of information sent to the display screen. (See Chapter 3, "BASIC Input and Output.")

PRINT#

```
::= ?'PRINT# filenum [, recnum] [; expr [{; expr]} [;] ]

)PRINT# 1; W$(0,0,0), LEFT$(W$(0,0,1))
)PRINT# 10, 4755; A&+24, T&/43, R%
```

PRINT# writes information to files in the same way that PRINT writes information to the screen. Its syntax is the same as that described for the PRINT statement above. A list of expressions separated by commas follows, the file reference number (or record number, if included).

One line of text is written for each expression in the list. PRINT# automatically performs any necessary numeric to string type conversions (similar to the STR$ function) in order to transfer the information from the expressions to the file.

A PRINT# statement in which a specific record number is given starts writing information to the file at the beginning of the specified record.

You can use he SPC specification with PRINT# statements in the same way that it is used with PRINT statements.

PRINT# USING

```
::= ?'PRINT# filenum [, recnum] USING linenum'string'svar [;expr[{,expr}]] [;]
```

The PRINT# USING statement is the same as a PRINT# statement with a USING clause, used to control the format of information sent to a file. (See Chapter 3, "BASIC Iput and Output.").

PROGNAM$

```
::= PROGNAM$
```

PROGNAM$ returns as a string the filename of the current program, preceded by the two characters 7/. The function MID$(PROGNAM$,2) will return just the program name. This name is set with the SAVE or SAVE AS filename variation of the SAVE statement.

PUT#

```
::= PUT# filenum[,length[,recnum];stvar
```

The PUT# statement transfers **n** bytes from the structure array to the random-access file given by **filenum** into the record given by **recnum**. The value of **n** is equal to the record size (set when the file is opened), unless a **length** parameter is given.

Structures are arrays with an element length of 1 byte. They are defined by the DIM statement, just like any other array. They can be used like single-integer array elements in numeric expressions, that is, as short positive integers from 0 through 255. Like normal arrays, structures can have multiple dimensions.

PUT# is used with a random-access file of a defined record length. It will transfer the smaller of the record size or the left dimension size minus the left subscipt plus 1, unless the record size is equal to 1. When the record size is 1, the length parameter is used. GET and PUT# always begin data transfers with the first byte of a record when the **recnum** option is used. When the **recnum** option is omitted, the transfer begins at the current file position, but the transfer length is limited to the remaining fraction of the current record. GET and PUT# will transfer data to and from DATA, BIN, or TXT files.

The PUT# statement limits the size of a transfer to the leftmost dimension of a structure or the maximum record size (32767 bytes).

QUIT

```
::= QUIT [filepath]
```

QUIT terminates GS BASIC, returning to the control program that initiated it. If GS BASIC was itself the control program initiated by ProDOS 16, control returns to the Apple IIGS Start Next Program default selector.

If the optional **filepath** is entered, the operating system will attempt to initiate a program with the given name; prefix 0 will apply if a partial pathname is used.

R.STACK%, R.STACK@, and R.STACK&

```
::= R.STACK%(ubexpr)
::= R.STACK@(ubexpr)
::= R.STACK&(ubexpr)
```

The R.STACK functions return data from the CALL return stack, a 32-byte buffer. The CALL and CALL% statements save up to 32 bytes or 16 words of results passed from a toolbox function in the return stack buffer. (See the description of CALL% in this chapter.) The unsigned byte expression is a word offset into the 32-byte return stack buffer; it must be in the range of 0 through 16 for R.STACK%, 0 through 15 for R.STACK@, and 0 through 13 for R.STACK&.

R.STACK@ returns a double word (32 bits) from the return stack as a signed integer number, R.STACK% returns an integer (16 bits) from the return stack as a signed integer number, and R.STACK& returns a long integer (64 bits) from the return stack as a signed integer number.

R.STACK%(0) will return a word containing the number of words left in the R.STACK by the last CALL statement execution. R.STACK%(1) will return the word that was at the top of stack after return from the toolbox function. The return stack is cleared upon return from every execution of a CALL or CALL% (even if its result word count is zero), so you must remove any return stack results before the next use of CALL or CALL% in your program.

RANDOMIZE

```
::= RANDOMIZE aexpr
```

RANDOMIZE reseeds the random number generator. The **aexpr** is the new seed and must be a positive integer in the range of 1 through 2^{31}-2. The expression may be derived from the reserved variable SECONDS@ after a TIMER ON statement has been executed. SECONDS@ will seed the random-number generator with the number of seconds since midnight (0 through 86399).

READ

```
::= READ var [{,var}]
```

```
)2001 READ Odyssey$,Wine%,Dark&,C
```

READ assigns the variable in its list values taken from elements in the program's DATA statement list. The following rules apply when you are assigning values to string:

- If the first nonblank character is either a quotation mark (") or an apostrophe ('), that character is the ending delimiter and all characters up to (but not including) the next occurrence of the delimiter are the value assigned to the string variable.
- If the first nonblank character is not a quotation mark or an apostrophe, all characters, including the first nonblank one, up to (but not including) the next comma are the value assigned to the string variable.

If a READ statement attempts to assign a string data element value to an arithmetic variable, the

```
?SYNTAX ERROR
```

message appears when the incorrect value type is assigned.

❖ *Note:* the rules for assignment of string variables using READ differ from those using INPUT.

Variables are assigned values of zero or null string (depending on the variable's type) when any of the following conditions are met:

- A comma is the first nonspace character following the reserved word DATA.
- There is no data element between two commas.

- The last character in a DATA statement is a comma (when the comma is being read as a data element).

READ#

```
::= READ# filenum [, recnum] [; var[{,var}] ]

)READ# 7; Pip1, Pip2

)READ# 8, 54; Twelve%, Strong&(2)
```

READ# gets information from a data file, specified by a file reference number. An optional record number may be included to specify a particular record in a random-access file to begin reading. A variable list following the file reference number (and optional record number, if included) defines where to put the information being read. (See the section "Accessing Data Files" in chapter 5 for the conversion limits of the READ# statement.)

The following table defines the conversion limits of the READ# statement:

Variable to Data Field Type		Result
Real to:		
	Real	OK
	Double Real	OK Possible loss of accuracy
	Integer	OK
	Double Integer	OK; Possible loss of accuracy
	Long Integer	OK; Possible loss of accuracy
	String	TYPE MISMATCH ERROR
Double real to:		
	Real	OK
	Double Real	OK
	Integer	OK
	Double Integer	OK;
	Long Integer	OK; Possible loss of accuracy
	string	TYPE MISMATCH ERROR
Integer to:		
	Real	OK in the range of +-32K, else OVERFLOW
	Double Real	OK in the range of ±32K, else OVERFLOW
	Integer	OK
	Double Integer	OK in the range of ±32K, else OVERFLOW
	Long Integer	OK in the range of ±32K, else OVERFLOW
	String	TYPE MISMTCH ERROR

Double integer to:

Real	OK in the range of ±2E+9, else OVERFLOW
Double Real	OK in the range of ±2E+9, else OVERFLOW
Integer	OK
Double Integer	OK
Long Integer	OK inThe range of ±2E+9, else OVERFLOW
Tring	TYPE MISMATCH ERROR

Long integer to:

Real	?OVERFLOW ERROR if more than ±9E+18
DoubleReal	?OVERFLOW ERROR if more than ±9E+18
Integer	OK
Double Integer	OK
Long Integer	OK
String	TYPE MISMATCH ERROR

String to:

Real	TYPE MISMATCH ERROR
Double Real	TYPE MISMATCH ERROR
Integer	TYPE MISMATCH ERROR
DoubleInteger	TYPE MISMATCH ERROR
Long Integer	TYPE MISMATCH ERROR
String	OK

REALS

::= [+'-] {**digit**}[.{**digit**}] E[+'-]{**digit**}
::= [+'-] {**digit**}.[{**digit**}] [E[+'-]{**digit**}]
::= [+'-][{**digit**}].{**digit**} [E[+'-]{**digit**}]

A real is any positive or negative number, and it can have a fractional part. A numeric constant with a decimal point is always of type real, even if it has only zeros to the right of the decimal point. However, all real constants must have either a decimal point, an exponent, or both; otherwise, the constant is considered an integer.

Reals whose absolute values are greater than or equal to $10^{-showdigits}$ and less than $10^{showdigits}$ are printed in conventional fixed-point notation. For example, if SHOWDIGITS is set at the default of 7, then 1, +1, -1., 3.14, 999.999, and -0.00002 are all real numbers that will be printed in fixed-point notation.

A real may also be expressed in scientific notation, such as 3.3E2, -3.3E4, 3.3E-4, or -3.3E-3. The real number 5.3E12, for example, is equal to 5.3 times 10 raised to the twelfth power.

Here are examples of conventional notation vursus scientific notation.

Conventional notation	**Scientific (E) notation**
300	3E2 = 3*(10^2)
320	3.2E2 = 3.2*(10^2)

.44	4.4E-1 = 4.4*(10^-1)
–.033	–3.3E-2 = 3.3*(10^-2)
1000000000000	1E12 = 1*(10^12)

Single-precision reals can represent numbers with seven significant digits within the following ranges -INF,-3.4E+38 through -1.5E-45, zero, 1.5E-45 through +3.4E+38,+INF.

Double-precision reals can represent numbers with 15 significant digits within a much larger ranges as follows: -INF,-1.7E+308 through -5.0E-324, zero, +5.0E-324 through +1.7E+308,+INF.

All mathematical computations involving single- or double-precision variables, constants, or long integers are actually done with extended precision before storing the result. Extended precision can represent numbers with 19 significant digits and a numeric range of -1.1E+4932 through 1.1E+4932. The SANE mathematical engine also provides for representation of plus and minus infinity, and various NaN results generated by impossible operations, such as dividing zero by zero or tyring to obtain the square root of a negative number.

REC

```
:= REC(filenum)
```

REC returns the current record number of the file specified by the value of the arithmetic expression following the reserved word REC.

If you use the INPUT# or READ# statements to access the catalog of a directory, REC returns the number of the line currently being accessed.

REC has the same error conditions as the TYP function.

REM

::= REM **anything**

```
)100 REM This can be a lifesaver.
```

The reserved word REM must be the first thing in a remark statement or the statement will not be treated as a remark. REM statements must not exceed 250 characters in length. If you comment your programs heavily, use several REM statements in successive lines rather than using one very long remark.

RENAME

```
::= RENAME pathname1, pathname2 [, FILTYP=TXT'SRC'BDF'ubexpr]

)RENAME /Floppy2/Animals/Dogs,/Floppy2/Animals/Pigs
```

RENAME is used to change the names of volumes, subdirectories, and local files. RENAME's argument list is composed of the old pathname, followed by a comma, followed by the new pathname, optionally followed by a file type specification. When the file type specification is included, the FILETYPE of the renamed file will be changed after a successful rename operation. You can change just the type of file by renaming it with the same name, but with a new file type specification.

You cannot use the RENAME statement to create a file or subdirectory, only to rename an existing one. Use the CREATE statement to make new files and subdirectories.

A local filename or subdirectory cannot be changed to another volume name or subdirectory; attempting this will result in a duplicate filename error.

RENUM

```
::= RENUM [newlinenum][, [increment] [,linenum1[-linenum2]]]
```

The RENUM command will renumber the lines of the currently loaded program. After renumbering your program, you must save it if you want to keep the newly renumbered program. RENUM first clears all the arrays, variables, and string data from the user data segment, as if you typed CLEAR.

The user data segment is then used to build the temporary renumber data tables during renumbering. If the user data segment is not large enough to accomplish renumbering, an

```
?OUT OF MEMORY ERROR
```

message will be displayed, and the program will remain unchanged. RENUM can only be used as a command, it cannot appear in a statement of a program.

RENUM will renumber, and if necessary resequence, your program. The four renumbering parameters can be used in various ways. If you don't enter any parameters with the command, the **newlinenum** and **increment** paramerers default to a value of 10. BASIC will renumber the entire program, assigning the first line number 10 with a line the number increment of 10.

RENUM 1000 means renumber the entire program in increments of 10 but assign the first line the number 1000. RENUM ,50 means renumber the entire program in increments of 50, assigning the first line the number 10.

After RENUM builds a table of the old line numbers and the corresponding new line numbers, the program is searched for all references to line numbers in GOTO, GOSUB, ON ... GOTO, THEN, ELSE, and similar statements, and the old references are checked to verify that all lines can be correctly modified for renumbering without losing any information.

Information might be lost because line numbers are tokenized (converted from characters into binary) and tokenization creates 1, 2, or 3 bytes for a tokenized line number. BASIC may have to increase the size of a program line to renumber it because the tokenized form of a line number reference may grow from 2 to 3 bytes. Because a line is limited to 255 bytes, it is possible to create one that cannot be renumbered later.

RENUM checks all lines that must have their references changed because of renumbering to ensure that all the lines can be renumbered without losing information in any line of the program. If RENUM findsa line that cannot be renumbered, the message

```
?LINE TOO LONG IN LINE nnnn
```

will appear, where **nnnn** is the number of that line.

After all lines are checked, the program is actually renumbered. If for any reason RENUM cannot renumber everything correctly, the program remains unchanged, and a message is displayed.

The last two parameters are only used when you want to renumber or move (by giving it new line numbers) a portion of the program. The **linenum1-linenum2** parameters define a range of lines within the program that are to be renumbered. Both **linenum1** and **linenum2** refer to the existing line numbers of the program before renumbering, but they need not refer to specific lines.

RENUM searches for **linenum1** and **linenum2** and uses that line or the first line with a larger line number if the specific line number given does not exist. When you specify **linenum1** without **-linenum2**, RENUM uses the last line number in the program for **linenum2**.

RENUM with **linenum1** and/or **linenum2** may cause the lines of the range to be moved within the program. This occurs when the **newlinenum** is less than the line number of the line prior to **linenum1** in the program or when **newlinenum** is greater than the line number of the first line after **linnum2**.

When a range of lines is renumbered in place, or moved to a new position, the new line numbers for the range cannot overlap the line numbers of any existing lines in the program outside the range. RENUM checks for this and issues the message

```
?CAN'T RENUMBER ERROR
```

when it occurs, leaving the program in memory unchanged.

REP$

```
:= REP$(sexpr,ubexpr)
```

REP$ returns a string of length **ubexpr** whose characters are all the first character **sexpr**. The **ubexpr** must be a number in the range of 0 through 255.

RESTORE

```
::= RESTORE [linenum|label]
```

RESTORE moves the data list pointer back to the beginning of the data list, allowing you to read the same data more than once. If the optional line number is given, the data read pointer is set to the beginning data item in the DATA statement at the given line number. If the line number given is not a DATA statement, an invalid line/label error occurs.

RESUME

```
::= RESUME [NEXT|COPY]
```

The RESUME statement, without either option, attempts to restart the statement that caused the most recent ON ERR or ON EXCEPTION event trap. RESUME without either option should only be used in an ON ERR processing routine. IIGS BASIC ignores RESUME statements it encounters until an error occurs. If you try to resume in immediate mode, an illegal direct error results.

RESUME NEXT may be used in place of RESUME in an ON ERR processing routine to execute the statement after the one that caused the error.

RESUME COPY is used to return control to the COPY statement for implementation of single-drive file-copy programs, (see the description of COPY in this chapter).

RETURN

```
::= RETURN
::= RETURN 0
```

When executing a RETURN statement, BASIC removes one pointer from the top of the for-gosub stack and branches to the statement indicated by the pointer. This is the statement immediately following the most recently executed GOSUB statement, unless a POP statement has been executed since the most recent GOSUB was encountered.

If BASIC attempts to execute one more RETURN statement than it has pointers on the program stack, a RETURN without GOSUB error occurs.

If you do not want to return to the statement following a GOSUB, possibly due to an error, you may return to another specific line in your program by using the sequence POP: GOTO **linenum|label**.

RETURN 0 is a special case of RETURN used at the end of the event-processing routines defined by EVENTDEF and MENUDEF for use with TASKPOLL. RETURN 0 resets the current statement to the one given by the pointer from the top entry on the for-gosub stack, but then executes an assembler RTL instruction to return to the event-dispatching code.

Warning:

RETURN 0 should never be used except in conjunction with TASKPOLL and the event-handling routines it calls. Misuse of RETURN 0 will crash the system.

RIGHT$

```
::= RIGHT$(sexpr, ubexpr)
```

```
)PRINT RIGHT$("Appleskin" + "Ware", 8)
skinWare
)B$=RIGHT$("Fruitbat", 3) : PRINT B$
bat
```

RIGHT$ returns a string of specified length composed of the rightmost characters of the given string expression. The length is given by the unsigned byte expression, and it must be in the range of 0 through 255, otherwise, an illegal quantity error will occur.

ROUND

```
::= ROUND(aexpr)
```

The ROUND function returns the integral value nearest the value of the **aexpr**, according to the rounding direction of the SANE settings. ROUND should be used in place of the common INT(**aexpr**+.5) because it will return a result consistent with the other capabilities of SANE.

RND

```
::= RND(aexpr)
```

```
)PRINT RND(8)
.830965
)
```

The RND function returns a random, real positive number less than 1.

RND generates a new random number each time it is used if the argument value is greater than zero.

RUN

```
::= RUN pathname[, linenum|label]
::= RUN [linenum|label]
```

```
)RUN
)RUN 205
)RUN Marathon
)RUN Assets, 7254
```

RUN is used to start running a program. When a RUN statement is entered, BASIC clears all variables, closes all open files except executing text files, and begins to execute the program in memory beginning with its smallest line number, or at the line number indicated. A program on disk can be run by following RUN with the program's pathname.

If you specify a nonexistent line number, an undefined statement error occurs. If the file you specify is not found after searching the disk, a file not found error occurs. If the file type of the file is not a IIGS BASIC program file type, a file type error occurs.

SAVE

```
::= SAVE [pathname]
::= SAVE AS [pathname]
```

SAVE writes a copy of the program currently in memory to a disk file. You can specify the pathname to be used by SAVE. If you have previously loaded, run, or chained the current program, the pathname is not required, and IIGS BASIC will save the program back to disk using its original name.

The SAVE AS variation of SAVE is used to set and display the current name; SAVE AS **pathname** will set the name *without* saving the program to disk. SAVE AS without the pathname will display the name on the next line of the screen display.

BASIC saves the program name and its associated prefix in memory (the prefix is set into prefix 7). SAVE without a pathname is the equivalent of SAVE 7/**program-name.**

Prefix 7 is set equal to prefix 1 (application prefix) during IIGS BASIC startup. Prefix 7 is updated whenever a full or partial pathname is used in a CHAIN, LOAD, RUN, SAVE AS, or SAVE statement.

If there is already a BASIC program with the same pathname on the disk, it will be overwritten and lost. If a locked BASIC program with the same name is on the disk, you will get a file locked error. If a file on the disk having the specified name is not a BASIC program, a file type error occurs, and the file is not saved.

SCALB

```
::= SCALB(siexpr,aexpr)
```

SCALB scales the arithmetic expression by $2^{\textbf{siexpr}}$, effectively returning the operand shifted left or right **siexpr** binary places. LOGB is related to SCALB, returning the **siexpr** for a given **aexpr**.

SCALE

```
::= SCALE(iexpr, aexpr)
```

```
)A&=12345678901234567
)PRINT USING "$$20&#.##";SCALE(-2,A&)
$123,456,789,012,345.67
```

SCALE is used in conjunction with PRINT USING to shift the decimal point of a displayed value to the left or the right. SCALE uses two arithmetic expressions as arguments. The first argument defines the number of places to the right that the decimal point should be moved. The second argument is the actual numeric value to be output.

The resulting exponent of the value must be between -4951 and +4532, or an illegal quantity error occurs.

scispec

```
::= [+˙-] [scipart] [fracpart] exp
```

```
)PRINT USING "+#.4#4E"; 3.1415926
+3.1416E+00
)PRINT USING "+.4#4E"; 3.1415926
+.3142E+01
```

The scientific notation specification (scispec) formats numeric output in scientific notation. The scispec is simpler than the fixspec, having either one digit or none to the left of the decimal point.

The # characters, either stated explicitly or by a repeat factor, define the number of digits to the right of the decimal point. The exponent position is defined with the letter E, and you can use a repeat factor.

Either three, four, five, or six character positions must be allowed for the exponent; four positions are adequate for all single-precision real variables, and five are adequate for all double-precision real variables. Six character positions are allowed for printing expressions that may have very large exponents.

When the spec calls for one digit position to the left of the decimal point, the first significant digit of the value is placed there; when there is no digit position to the left of the decimal point, the most significant digit is placed to the right of the decimal point. In either case, the exponent is then calculated to make the displayed value correct.

SECONDS@

```
::= SECONDS@
```

SECONDS@ is a reserved variable that returns the value of counter maintained by the TIMER ON statement. SECONDS@ will return a positive number in the range of -1 through 86400. The value 0 is returned if TIMER ON has not been executed. If TIMER OFF mode is currently in effect, the value of the timer will return an unchanging number.

SECONDS@ will remain a zero until the first TIMER ON statement is executed. Due to the presence of numerous interrupt sources in the Apple IIGS, many of which have higher priority than the 1-second clock interrupt, the SECONDS@ value is not always exact. However, SECONDS@ will always be exact immediately after execution of the TIMER ON statement. TIMER ON may be used as often as needed during a program.

SET

```
::= SET(stvar, [aexpr1]) = expr
::= SET(stvar, [aexpr1]) =^ sexpr
::= SET(stvar, [aexpr1]) =*divar[,length]
```

The SET statement allows storing a variable or expression result into a structure array. The optional arithmetic expression, **aexpr**, is the size of the destination field and must result in a positive number in the range of 1 to the leftmost dimension of the **stvar** (if it is used).

SET assigns the result of the expression to the structure array according to the type of the expression result. The expression result will be an integer only if the expression evaluation was able to complete all operations using integer calculations or the expression is just an integer variable.

The size of the result controls the field size stored if the optional length parameter, **aexpr1**, is not given. The size for various variable and expression results is show in the table below. If the length parameter is given and the expression result (right of the equal sign) is smaller than the field size, just the size of the result is stored, and any extra bytes in the field are left unchanged. The result is always stored low byte at the lowest address, which corresponds to lowest structure array subscripts.

The default field sizes for an expression that is just a variable reference are as follows:

Single-precision real variable	4 bytes
Double-precision real variable	8 bytes
Single-integer variable	2 bytes
Double-integer variable	4 bytes
Long-integer variable	8 bytes
String variable	Length of the string(0..255 bytes)

An actual expression can return a result that is single or double integer or an extended precision real result. When the expression result is an extended-precision real, the result will be converted and stored as a double-precision real if the length is given as 8 bytes, as a single-precision real if the length is 4 bytes, or left as an extended precision real if the length is 10 bytes.

The second form of the SET statement is used to store a BASIC string into a structure array as a counted string (sometimes refered to as a Pascal string). The expression must be a string expression. This conversion will store LEN(**sexpr**)+**sexpr** in the structure array.

The third form of the SET allows two types of direct memory assignment to a structure array. When the optional comma and length expression are omitted, the value of the double-integer variable is used as the memory address of a 1-byte count, followed by 0 to 255 characters of string data that are assigned to the field in the structure array. If the comma and length expression are present, the address used is the address of length bytes of data. The count byte or the length parameter may be zero, and the length may be up to 32767.

SGN

```
::= SGN(aexpr)
```

```
)PRINT SGN(-234)
-1
)PRINT SGN(2496+234)
1
)PRINT SGN(5E4-5E4)
0
)
```

SGN returns -1 if the argument value is negative, 0 if the value of the argument equals 0, and 1 if the argument value is positive.

SHOWDIGITS

```
:= SHOWDIGITS
```

SHOWDIGITS is a modifiable reserved variable that controls how many significant digits are output by the PRINT statement. The default value of SHOWDIGITS is 7, the number of significant digits in a single-precision real number. SHOWDIGITS can be set to integer values in the range of 2 through 28 with the LET statement. SHOWDIGITS does not influence the behavior of PRINT USING or any other statement. (See the description of PRINT in this chapter.)

SIN

```
::= SIN(aexpr)
```

```
)PRINT SIN(2.718)
.411038
)
```

SIN returns the sine of an angle given in radians.

SPACE$

```
:= SPACE$(ubexpr)
```

SPACE$ returns a string of spaces in the length given by **ubexpr**. The unsigned expression must be a number in the range of 0 through 255.

SPC

```
::= SPC(aexpr)
```

```
)PRINT "A"; SPC(1); "B"; SPC(2); "C"
A B C
)PRINT "D"; SPC(5); "E"; SPC(5); "F"
D E F
)SPC(250)SPC(139)SPC(255)
```

SPC is used in PRINT statements to define (by the expression enclosed in parentheses) the number of spaces to be inserted after the last printed character.

Each SPC statement is limited to a maximum value of 255, but you can insert as many spaces as you want by stringing together a series of SPC statements.

SQR

```
::= SQR(aexpr)
```

```
)PRINT SQR(3^2+4^2)
5
)
```

SQR returns the positive square root of the argument value.

STEP

```
::= STEP aexpr
```

```
)2000 FOR Farenheit=1 to 451 STEP 3 : NEXT
)2005 PRINT "Fire!!!"
)87 FOR Counter=10 to -10 STEP -1. . .NEXT Counter
```

STEP allows you to increment (or decrement) the control variable of a FOR ... NEXT loop (described earlier) by integer steps other than 1. If a negative value is specified in a STEP clause, the loop counts backwards. If a positive value is specified, the loop runs forward.

The FOR statement coerces the value of the STEP expression to the type of the control variable. For example:

```
FOR I%=11 to 33 STEP 1.66
```

will execute with a step value of 2.00 because the step expression is treated as if you had entered STEP CONV%(1.66). The CONV% function will round the argument to an integral (the nearest whole number), and then convert it to an integer.

STOP

```
::= STOP
```

STOP halts execution of a program, terminates any executing text file, returns BASIC to immediate execution, resets the output file to .CONSOLE, and redisplays the prompt character. STOP displays a message, such as

```
?PROGRAM INTERRUPTED 8712
```

where 8712 is the line number of the program line containing the STOP statement. The program in memory is not altered in any way. STOP has no options associated with it.

STR$

```
::= STR$(aexpr)
```

```
)PRINT STR$(25/3)
8.33333
)PRINT STR$(100000000000)+"More"
1E+11More
)
```

STR$ evaluates a given arithmetic expression and returns the value as a string.

Strings

A string is a sequence of characters. String variable names must end with a dollar sign ($). Strings can contain from 0 characters (the null string) to 255 characters. The number of characters in a string is referred to as its length. Strings are not fixed in length, but may grow or shrink as necessary.

When a program is run, all string variables initially contain the null string.

SUB$

```
::= SUB$(svar, ubexpr [,ubexpr]) = sexpr
```

```
)F$="Hardware" : SUB$(F$,1)="Soft" : PRINT F$
Software
)F$="Hardware" : B$="Soft" : SUB$(F$,1,2)=B$ : PRINT F$
Sordware
)
```

SUB$ lets you replace any part of a string with a specified substring. The string to be changed can be any string variable, and the substring may be the value of any string expression. You must specify the first character in the string to be changed by following that string with an unsigned arithmetic expression in the range of 1 through 255.

You can optionally include a second unsigned arithmetic expression to specify the number of characters in the substring to replace characters in the original string.

SWAP

```
::= SWAP var1, var2
```

```
)SWAP Tick,Tock
)SWAP Old$,New$
```

SWAP exchanges the value stored in one variable for the value stored in another. You can use string, integer, and real variables with SWAP, but both variables must be of the same type.

TAB

```
::= TAB(aexpr)
```

```
)PRINT "Great"; TAB(8); 347
Great 347
)PRINT "Underhanded"; TAB(8); 553
Underhanded553
```

TAB is used in PRINT statements to define the number of spaces from the left margin of the text window to begin printing text. If you specify an expression that is less than the number of the current print position, no spaces will be inserted before the next character to be printed.

TAN

```
::= TAN(aexpr)
```

```
)PRINT TAN(3.141)
-5.92653E-04
)
```

TAN returns the tangent of an angle given in radians.

TEN

```
::= TEN(sexpr)
```

```
)PRINT TEN("30C")
)PRINT TEN(" $E02033")
```

TEN returns the decimal (base 10) equivalent of the hex digits of the given string expression. The value returned will be in the range of a double integer. The expression may contain leading spaces followed by an optional dollar sign character, but the next **eight** or fewer characters of the string result must represent a hexadecimal number; if not, an illegal quantity error occurs.

TASKPOLL

```
::= TASKPOLL INIT iexpr1,iexpr2
::= TASKPOLL ON
::= TASKPOLL OFF
```

The TASKPOLL statement enables and activates Window Manager and Menu Manager event polling through the Task Master function of the Window Manager. This statement is described in more detail in Chapter 7, "Advanced Topics".

Prior to executing TASKPOLL INIT, you must load and properly initialize the Window Manager, the Menu Manager, the Desk Manager, the Control Manager, and any other tool set you are using, and define all the required data structures with CALL or CALL%. The tool set functions required for this complex programming task are fully documented in the *Apple IIGS Toolbox Reference* manual. In addition, you must define for IIGS BASIC the event-handling routines in your program with the EVENTDEF and MENUDEF statements, described in this chapter and in Chapter 7, "Advanced Topics."

Warning

If you haven't read and understood the Toolbox documents on using Task Master and all the related tool sets, don't use TASKPOLL in your programs.

The **iexpr1** in TASKPOLL INIT is a double-integer variable stored in the TaskMask field of the TaskRec used by IIGS BASIC when calling Task Master. The second integer expression, **iexpr2**, is used as the EventMask in the Task Master call parameters. It is stored in the word preceding the TaskRec and can be examined with TASREC%(0).

You must execute TASKPOLL INIT before executing TASKPOLL ON; if you do not, you will see the message

```
?TASKPOLL INIT &/ WINDSTARTUP ERROR
```

TASKPOLL INIT also checks to be sure that the Window Manager is active when TASKPOLL INIT is executed, and if it is not, TASKPOLL ON will generate the above message, even if TASKPOLL INIT has set the TaskMask and EventMask.

The proper values to use for the TaskMask and EventMask are defined in the *Toolbox Reference* manual in the Window Manager and Event Manager chapters. The exact values you select will depend on how you decide to use Task Master.

EVENTDEF and MENUDEF provide the linkage between specific Task Master events and your program. IIGS BASIC supports 64 events (Task Master defines 29 as of March 87) plus up to 128 menu items. The menu items are not dispatched individually, but are returned as the wInSpecial event.

TASKPOLL ON enables actual Task Master polling, and TASKPOLL OFF disables Task Master polling. When polling is enabled with TASKPOLL ON, IIGS BASIC calls Task Master just before executing every statement. Task Master polling has lower priority than the ON TIMER, ON BREAK, and ON KBD events, all of which are checked before polling Task Master.

When Task Master returns a null event, the current statement is executed. If an event is returned by Task Master, it is dispatched through the EVENTDEF dispatch table, except for the wInMenuBar events, which are dispatched through the MENUDEF dispatch table using the item identification number returned by MenuSelect. If the selected entry in the appropriate dispatch table is zero, the event is discarded and program execution continues.

When the event or menu dispatch entry in the EVENTDEF and MENUDEF tables defines line number, an effective GOSUB **linnum** statement is executed. The event-handling routine in your BASIC program processes the event, and then executes a RETURN 0 to return control to the next program statement. Task Master polling is suspended until the RETURN 0 is executed, so subsequent events are held in the event queue until BASIC completes processing for the current one.

TASKREC% and TASKREC@

```
::= TASKREC%(ubexpr)
::= TASKREC@(ubexpr)
```

The TASKREC functions return a single- or double-integer result from the Task Master TaskRec. The unsigned byte expression is a word offset into the TaskRec, which is an internal BASIC data structure used when calling Task Master.

The TaskRec is an extension of the Window Manager task record structure. The first word of TaskRec is the Event Manager mask used by IIGS BASIC when calling Task Master. The remainder is the same as the Task Master task record, as defined under Using Task Master in the Window Manager chapter of the *Toolbox Reference* manual.

TEXT

```
::= TEXT
```

The TEXT statement sets the display screen to the usual full-screen text mode. It clears any other text or graphics mode in use and displays a prompt and the cursor at the left margin of the next line.

TEXTPORT

```
::= TEXTPORT ubexpr1, ubexpr2 TO ubexpr3, ubexpr4

)TEXTPORT 37,9 TO 44,16
```

TEXTPORT allows you to set the position and size of the textport, a rectangle within the total screen area where BASIC may display text.

The first pair of numbers specifies the horizontal and vertical coordinates of the upper-left corner of the textport, and the second pair specifies the coordinates of the lower-right corner. The example above will create a textport eight columns wide and eight screen lines high, in the center of your screen. When a TEXTPORT statement is executed, the cursor moves to the upper left corner of the specified textport.

TEXTPORT statement coordinates may be specified by any arithmetic expression. Each of the four expressions must have a value within the range of 0 through 255, or an

```
?ILLEGAL QUANTITY ERROR
```

message will be displayed. If your values would make the textport larger than the maximum allowed screen size (24 x 80), the textport is truncated to fit.

TIME$

```
::= TIME$ ubexpr1, ubexpr2, ubexpr3
```

You can set the Apple IIGS clock time, without changing the current date, with the TIME$ statement. The hour is set from **ubexpr1** and must be in the range of 0 through 23, the minute is set from **ubexpr2**, and the second from **ubexpr3.** Both the minutes and seconds must be in the range of 0 through 59.

TIME

```
::= TIME( ubexpr)
```

The TIME function is used to read the Apple IIGS clock time fields as numbers rather than as the string returned by TIME$. The **ubexpr** must be in the range of0 through 3. The TIME function must be called with a zero argument to actually update the values returned for the other arguments. The result returned for argument zero is the hour in 24-hour format (0 through 23).

The requirement that the function zero be called first protects you from having the hour, minute, or second change between calls for the other results. This problem is commonly known as **the clock rollover problem.** If TIME(0) is not called immediately prior to using the other parameters, the function results will reflect the time as of the previous TIME(0) call. You should not call TIME(0) a second time until you have retrieved all the other results into your variables.

Function	Result	
TIME(0)	Hour (0 through 23)	reads the clock
TIME(1)	Hour (0 through 23)	doesn't read the clock
TIME(2)	Minute (0 through 59)	
TIME(3)	Second (0 through 59)	

TIME$

```
::= TIME$
::= TIME$ ubexpr1, ubexpr2, ubexpr3
```

TIME$ is both a reserved variable (first form) that returns the current time as a string, and a statement (second form) that sets the Apple IIGS clock time to the hour given by **ubexpr1** (which must be in the range of 0 through 23), the minute given by **ubexpr2**, and the second given by **ubexpr3.** Both the minutes and seconds must be in the range of 0 through 59. The current date setting is not disturbed by changing the time.

TIMER ON and TIMER OFF

```
::= TIMER ON
::= TIMER OFF
```

TIMER ON reads the time from Apple IIGS clock, calculates "seconds since midnight" (a number from 0 to 86399), and stores the number in a counter that is updated once a second. The counter is maintained by using the 1-second clock interrupt. After TIMER ON is executed, a program may initiate a one-shot interval timer using the ON TIMER statement. TIMER OFF disables the 1-second clock interrupt and thus freezes the SECONDS@ counter.

Because of the low priority of the 1-second interrupt and other factors in the system, it is possible for the counter to miss an interrupt an not reflect the actual number of seconds since midnight.

TRACE

```
::= TRACE
::= TRACE TO #filenum
```

```
)TRACE
)RUN
```

TRACE prints a # followed by the number of each line of a program as it executes. The optional TO clause directs the trace information to a file that has been opened with the OPEN statement. The file can be either a character device or a disk file. If it is a disk file, the program will slow down whenever the trace information is written to the drive, and may itself cause a disk full error.

A very useful means of tracing a program is to send the trace information to a printer, a RAM disk file, or another computer display through a parallel or serial interface. When the trace information is directed to a file, the program could also send additional diagnostic information to the file with PRINT# statements.

TRACE is switched off by rebooting, a LOAD **pathname** statement, a RUN **pathname** statement, or by typing NOTRACE. CHAIN or RUN statements do not cancel TRACE.

TYP

```
::= TYP(filenum)
```

```
)ON TYP(3) GOSUB 1000,1200,1400,1600,1800,2000
```

TYP is used to determine what type of data will be read from a BASIC data file (FILTYP=BDF) on the next access to that file. The argument to the function can be any arithmetic expression, but its value must specify a particular file reference number. The number returned by the TYP function denotes what type of data will next be read from the specified file.

For a BASIC data file, TYP returns the following values:

0	end of file
1	not returned by
2	next datum is integer
3	next datum is double integer
4	next datum is long integer
5	next datum is single Real
6	next datum is double Real
7	next datum is string

If there are no more data items in the file the value 0 is returned. If the type of the file is not a BASIC Data File, a file type error occurs.

TYPE

```
::= TYPE pathname [TO # filenum][,width]
```

The TYPE command will attempt to open a SRC or TXT file, read lines of text terminated by carriage returns, and display the lines on the screen. If the optional **width** parameter is given, only the first width characters of each line are displayed. If the optional TO # **filenum** is used, an implied OUTPUT# **filenum** :TYPE : OUTPUT # 0 sequence is executed, sending the text lines to the open file.

UBOUND

```
::= UBOUND(array-name[()][,dim-number])
```

UBOUND returns the upper bound of the dimensions of an array. The dimension parameter is an optional number that is used for multidimensional arrays. It specifies which dimension of the array to test. The upper bound of an array is the largest possible subscript for a given dimension. The lower bound of an array is always zero.

UCASE$

```
::= UCASE$(sexpr)
```

The UCASE$ function returns the string expression argument after shifting all the lowercase letters a through z up to the letters A through Z.

UIR%

```
::= UIR%(ubexpr)
```

The UIR% function returns the status information from the UIR after INPUT USING completes. The UIR function 0, exit type, is the index of the **tchar,** or termination keypress, that ended the input editing. It will be in the range 1of **n**, and it indicates the **tchar** in the IMAGE statement that was entered.

The UIR% function returns the following status results:

UIR%(0)	exit type
UIR%(1)	ASCII value of last keypress
UIR%(2)	masked modifier of last keypress
UIR%(3)	reentry type
UIR%(4)	last cursor x position
UIR%(5)	last cursor y position
UIR%(6)	last relative character position

The INPUT USING statement and the UIR% function are discussed in more detail in Chapter 7, "Advanced Topics."

UNLOCK

See the description of LOCK earlier in this chapter.

UNTIL

```
::= UNTIL
::= UNTIL lexpr

100 DO : statements … : UNTIL GH > 29
14000 WHILE : statements … :UNTIL GH >= 299
550 DO : statements : WHILE twgy > 11 : statements : UNTIL
```

UNTIL is used in conjunction with the DO and/or WHILE verbs to create various types of conditional loops. The verb UNTIL marks the end of the loop construct and can be used with or without the conditional expression.

If the conditional expression is omitted, UNTIL loops back to the most recently executed DO or WHILE statement (DO takes precedence if both DO and WHILE precede the UNTIL). When the logical expression is present, UNTIL will loop back if the expression if false (zero), and proceed to the next statement if the expression is true (nonzero).

❖ *Note:* The UNTIL logic test is loop back if false, but the WHILE test is skip if false.

Some other BASICs implement similar WHILE ... WEND or REPEAT ... UNTIL constructs that are easily duplicated with WHILE ... UNTIL by simply including or excluding the logical expression. The advantage of the WHILE ... UNTIL construct is the new combinations it allows:

```
WHILE lexpr : … statements … : UNTIL lexpr   (two tests)
WHILE lexpr : … statements … : UNTIL         (WHILE … WEND)
WHILE : … statements … : UNTIL lexpr         (REPEAT … UNTIL)
DO : … statements … : UNTIL lexpr            (DO … UNTIL)
WHILE : … statements … : UNTIL               (infinite loop)
DO : statements : WHILE lexpr :… statements …:UNTIL
DO : statements : WHILE lexpr :… statements …:UNTIL lexpr
```

The DO ... WHILE ... UNTIL construct *always* executes the statements before the WHILE and *conditionally* executes the statements after the WHILE. UNTIL examines the control stack for the WHILE information and if an UNTIL is executed without a prior DO or WHILE statement, an UNTIL without WHILE error will occur.

VAL

```
::= VAL(sexpr)
```

```
)PRINT 10 * VAL("1.3E4")
130000
)PRINT VAL("13"+"77")
1377
```

VAL evaluates a given string expression and returns the value as a real or an integer number.

If any character of the string expression value evaluated is not a legal numeric character (leading spaces are acceptable), a type mismatch error occurs.

If the absolute value of the number represented by the value of the string expression is greater than the range of a double-precision real number (approxamately 1.7E+308), an overflow error occurs (see the descriotion of reals in this chapter).

A string expression value containing more than 255 characters causes a string too long error.

VAR

```
::= VAR(stvar,vtype[,lgth])
::= VAR(address-expression,type[,lgth])
```

VAR is the inverse of the SET statement; it extracts a variable from a structure array (or memory) and becomes a variable of the **vtype**. VAR can be used wherever an **expr** can be used. VAR can also be the expression on the right side of a LET or SET assignment. **Vtype** is specified by using the numbers returned by the TYP function, as follows:

1	Result is an extended real
2	Result is an integer
3	Result is a double integer
4	Result is a long integer
5	Result is a single real
6	Result is a double real
7	Result is a string

The length parameter may be used with the integer types to specify a size smaller than the default size (2, 4, 8), and it must be used with the string type. For single integers, **lgth** may be 1 or 2; for double integers, **lgth** may be 1, 2, 3, or 4; and for long integers, it may be 1 through 8. For strings, **lgth** must be in the range of 1 through 255. When an integer is created from a reduced size, the result is always a positive number; that is, no sign extension is provided.

The second form of the VAR function allows a *multibyte peek* functionality. The interpreter distinguishes these two otherwise ambiguous definitions by the subscripted variable used as the first term of the first parameter. The expression is evaluated and converted (if necessary) into a double integer that is taken to be an address. (Expressions starting with subscripted variable names are not allowed in the second form of VAR; if the first term is a subscripted variable name, the first format is assumed, and the subscripted variable must be a structure array reference.

The type parameter still controls the type of what is peeked, but it allows peeking at an integer (type=2), a double integer (type=3) (dereferencing a pointer), a long integer, single or double reals, and text or P-strings. When a string is extracted, a P-string is assumed if type=7 and no length is given; that is the address is the address of a count byte followed by 1 to 255 characters.

Of special note is the variation

```
VAR(BASIC@(48)+x,3,3)
```

where *x* is a zero-page address. This peeks at a 3-byte pointer in IIGS BASIC's zero page. The type indicates that a double integer should be created; and the length says peek only 3 bytes. (Most of the GS BASIC zero-page pointers are 3 bytes rather than 4 bytes.)

VAR$

```
::= VAR$(aexpr[,ubexpr])
```

The VAR$ function creates a string variable from the counted string at the memory address given by the arithmetic expression, when the optional length expression is omitted. This function provides a means of extracting a string result returned by a tool set function. When the optional comma and length expression are present, the string result is taken beginning at the address given by **aexpr** for length bytes or until encountering a binary zero in memory. The unsigned byte expression, **ubexpr**, must have a value in the range of 0 through 255.

VARPTR and VARPTR$

```
::= VARPTR(name)
::= VARPTR$(svar)
```

The VARPTR function returns the address of the variable name given in parentheses. For string variables, VARPTR returns the address of the string descriptor, not the address of the string data. VARPTR$(**svar**) returns the address of the string data of a string variable. VARPTR$ will return a type missmatch error if a numeric variable is used. VARPTR and VARPTR$ will not create an array or a variable if an undefined array or variable is first referenced through VARPTR. VARPTR will return a variable error if an undefined variable or array element is referenced.

VARPTR$ will return a zero if a null string is referenced.

Variable types

There are six elementary variable types in Apple IIGS BASIC: single, double and long integers, single- and double-precision reals, and strings. The first five types represent numbers of various kinds, the last type represents sequences of characters.

The type of a variable is determined by the last character of its name: % for single integer, @ for double integer, & for long integer, # for double-precision real, and $ for string. In the absence of any of these special trailing characters, the variable type is considered to be a single-precision real by default.

The structure array is a variable type that is not a simple variable type. The last character of a structure name is the ! character, and it must always have a subscript. An element of a structure can be referenced like a numeric variable and will act like a single integer with a value of zero through 255.

Here are examples of names of the six variable types:

Name	Type
Length	single-precision real
HYPOTENUSE#	double-precision real
MYField![12]	structure element (byte integer)
Marbles7%	single integer
ADDRESS@	double integer
Light.Years&	long integer
Myname$	string

VOLUMES

```
::= VOLUMES
```

The VOLUMES command attempts to read the volume name for device names .D1 through .D9. A line per device display of .D**n /volumename freeblks** on the current console device or file is generated. If all devices known by the system have a volume mounted or a drive connected, a text message of

```
DRIVE EMPTY
```

or

```
DEVICE NOT CONNECTED
```

is substituted for the volume name and the free blocks count.

VPOS and HPOS

```
::= HPOS ' VPOS
```

The modifiable reserved variables VPOS and HPOS contain the vertical and horizontal positions, respectively, of the current print position. Changing their values will change the current print position (and the cursor's position). You can determine the position of the cursor by accessing the values of VPOS and HPOS.

Assigning values greater than the height of the text window to VPOS causes the cursor to move to the bottom screen line within the window. Assigning values greater than the width of the text window to HPOS causes the cursor to move to the right margin of the window. The value 0 is converted to the value 1. Assigning values outside the range of 0 through 255 to either VPOS or HPOS causes an illegal quantity error.

WHILE

```
::= WHILE
::= WHILE lexpr

100 WHILE : statements … : UNTIL GH > 29
14000 WHILE : statements … :UNTIL GH >= 299
550 DO : statements : WHILE twgy > 11 : statements : UNTIL
```

The WHLE verb is used with the UNTIL verb or with both the DO and UNTIL verbs to create various conditional loop constructs. The verb WHILE marks the beginning or midpoint of the loop construct and can be used with or without the conditional expression. Using WHILE between DO and UNTIL without a conditional expression is a meaningless (although valid) construct.

If the logical expression is omitted, WHILE behaves as if the expression were true. When the logical expression is present WHILE will execute the following statements if the expression is true (nonzero) and skip to the statement following the UNTIL if the expression if false (zero). The presence of a conditional expression in the matching UNTIL statement does not influence the behavior of WHILE.

WHILE searches forward in the program for a matching UNTIL and will display the message

```
WHILE w/o UNTIL ERROR
```

if an UNTIL is not present. Further examples and details of these constructs is found in the description of the UNTIL statement and in Chapter 4, "Controlling Program Execution."

WRITE#

```
::= WRITE# filenum [, recnum] [; expr[{, expr}] ]
```

```
)WRITE# 3; MAJOR%, MINOR%, XLOW
)WRITE# 4, 11; MAP(1,3 ,5,7,9)
```

WRITE# sequentially writes the value of each item in its expression list to a field in a specified data file. You can optionally follow the file reference number with a comma and an arithmetic expression specifying a record number at which to begin access. The list of expressions must follow the file reference number (or optional record number), and the expressions in the list must be separated by commas.

One item of data is written for each expression in the list. WRITE# performs no numeric-to-string type conversions while transferring information from the expressions to the file; it just writes a binary image of the data to the file, with a type byte in front.

A single integer is writtten as 3 bytes, a double integer as 5 bytes, a long integer as 9 bytes, a single real as 5 bytes, a double real as 9 bytes, and a string as the length of the string plus 2 bytes.

If a record number is specified, then the value of the first expression in the expression list is written to the first field in the specified record. Otherwise, records are accessed sequentially.

If there is not enough room left in a record to hold the next value, the field will be written in the next record. Note that writing data to a record causes any old data in the record to be lost. If an attempt is made to write a data field longer than the record length specified when the file was created, the message

```
?OUT OF DATA ERROR
```

is displayed.

Appendix A

ASCII Character Codes

ASCII is an acronym for American Standard Code for Information Interchange.

The range of standard ASCII codes extends from 0 to 127. Apple IIGS BASIC also treats the range of values 128 to 255 as valid codes, but they are not generated from the keyboard.

Legend:

DEC: ASCII code in decimal notation.

HEX: ASCII code in hexadecimal notation.

CHAR: ASCII mnemonics.

Table A-1
Control Characters

DEC	HEX	CHAR	Keyboard Action	Comments and Notes
0	00	Null	CONTROL-@	Null
1	01	SOH	CONTROL-A	
2	02	STX	CONTROL-B	
3	03	ETX	CONTROL-C	Halts execution
4	04	ET	CONTROL-D	
5	05	ENQ	CONTROL-E	
6	06	ACK	CONTROL-F	
7	07	BEL	CONTROL-G	Beeps speaker
8	08	BS	CONTROL-H	Backspace, (same as <--)
9	09	HT	CONTROL-I	Horizontal tab
10	0A	LF	CONTROL-J	Linefeed
11	0B	VT	CONTROL-K	Vertical tab
12	0C	FF	CONTROL-L	Formfeed
13	0D	CR	CONTROL-M	Carriage return (same as Return)
14	0E	SO	CONTROL-N	
15	0F	SI	CONTROL-O	
16	10	DLE	CONTROL-P	
17	11	DC1	CONTROL-Q	
18	12	DC2	CONTROL-R	
19	13	DC3	CONTROL-S	
20	14	DC4	CONTROL-T	
21	15	NAK	CONTROL-U	
22	16	SYN	CONTROL-V	
23	17	ETB	CONTROL-W	
24	18	CAN	CONTROL-X	Cancels line being edited
25	19	EM	CONTROL-Y	
26	1A	SUB	CONTROL-Z	
27	1B	ESC	Escape	Cursor control and editing
28	1C	FS	CONTROL-/	
29	1D	GS	CONTROL [	
30	1E	RS	CONTROL-^	
31	1F	US	CONTROL _	

Table A-2
Uppercase Letters, Numbers, and Symbols

DEC	HEX	CHAR	Keyboard	DEC	HEX	CHAR	Keyboard
32	20	SPACE	SPACEBAR	64	40	@	@
33	21	!	!	65	41	A	A
34	22	"	"	66	42	B	B
35	23	#	#	67	43	C	C
36	24	$	$	68	44	D	D
37	25	%	%	69	45	E	E
38	26	&	&	70	46	F	F
39	27	'	'	71	47	G	G
40	28	(	(	72	48	H	H
41	29	)	)	73	49	I	I
42	2A	*	*	74	4A	J	J
43	2B	+	+	75	4B	K	K
44	2C	,	,	76	4C	L	L
45	2D	-	-	77	4D	M	M
46	2E	.	.	78	4E	N	N
47	2F	/	/	79	4F	O	O
48	30	0	0	80	50	P	P
49	31	1	1	81	51	Q	Q
50	32	2	2	82	52	R	R
51	33	3	3	83	53	S	S
52	34	4	4	84	54	T	T
53	35	5	5	85	55	U	U
54	36	6	6	86	56	V	V
55	37	7	7	87	57	W	W
56	38	8	8	88	58	X	X
57	39	9	9	89	59	Y	Y
58	3A	:	:	90	5A	Z	Z
59	3B	;	;	91	5B	[	[
60	3C	<	<	92	5C	\	\
61	3D	=	=	93	5D	]	]
62	3E	>	>	94	5E	^	^
63	3F	?	?	95	5F	_	_

Table A-3
Lowercase Letters and Symbols

DEC	HEX	CHAR	Keyboard
96	60	`	`
97	61	a	a
98	62	b	b
99	63	c	c
100	64	d	d
101	65	e	e
102	66	f	f
103	67	g	g
104	68	h	h
105	69	i	i
106	6A	j	j
107	6B	k	k
108	6C	l	l
109	6D	m	m
110	6E	n	n
111	6F	o	o
112	70	p	p
113	71	q	q
114	72	r	r
115	73	s	s
116	74	t	t
117	75	u	u
118	76	v	v
119	77	w	w
120	78	x	x
121	79	y	y
122	7A	z	z
123	7B	{	{
124	7C	|	|
125	7D	}	}
126	7E	~	~
127	7F	none	DEL

Appendix B

Errors

When BASIC detects a program error during deferred execution, it checks to see if an ON ERR statement is in effect. If so, program execution jumps to the statement list following the reserved words ON ERR. If not, BASIC halts execution of the program, displays a brief error message, and displays the prompt and cursor. Variable values and the program text remain intact, but program execution can not continue. All program stacks and pending FOR/NEXT loops are discarded.

GS BASIC checks the context of various classes of reserved words during program entry, but does not check the exact syntax of each statement until it is executed. Following are some of the more common types of errors; namely grammatical or syntax errors, data content errors, memory management errors, and file I/O errors.

Grammatical or syntax errors include misspelled or misused reserved words, missing or incorrect punctuation characters, improper line numbers or labels, correct clauses or options in the wrong order, and missing clauses required by a specific variation of a statement.

Data-content errors are very common and may sometimes be reported by BASIC as syntax errors. Examples of this include using a valid line number or label that references a nonexistent line, procedure or function; using a valid numeric constant that is outside the allowable range required by its context, such as a subscript that is too large; using a real number in a context that automatically rounds it to an integral number or forces conversion to an integer; and using a string expression when a numeric one is required and vice versa.

Memory-management errors are often the result of incorrect design concepts, the most common of which is not assuming that all available memory will eventually be consumed by a program. Generally, it is wise to design software with fixed limits and overflow management strategies to avoid the inevitable

```
?OUT OF MEMORY ERROR
```

message and less frequent

```
?STACK OVERFLOW ERROR
```

message. Examples include allocating a string array with more elements that can be represented in a 64K-string data pool, and not expanding the user data segment with the CLEAR statement or expanding it too the limit of system memory.

File I/O errors can occur at any time in an otherwise perfect program. Examples include misspelled or invalid user-entered filenames, volume names, or pathnames; valid filenames for files with the wrong file types; attempting disk file operations on character (device) files; attempting operations on a file after it has been closed or before it has been opened; failures due to removal of a disk volume from its drive after opening files; running out of disk space or directory entries in a fixed size root directory; and attempting to use a restricted access file in the wrong mode.

Format of error messages

When an error occurs in an immediate execution statement, an error message is immediately displayed. For example:

```
)PRINT MID$(234)
?TYPE MISMATCH ERROR
)
```

When an error occurs in a deferred execution statement, an error message is displayed, complete with the line number of the erroneous statement. For example,

```
)10 PRINT MID$(234)
)RUN
?TYPE MISMATCH ERROR IN 10
)
```

Note, however, that the error message only identifies the line number of the statement causing the error; when a line contains multiple statements, you cannot always be certain which statement generated the error. Often, the statements in a line are sufficiently different that only one statement could be the source of a specific error message, but syntax errors can occur in almost every statement. You can always separate a multistatement line into separate lines to identify exactly which statement is generating an error.

Error Messages

The following is an alphabetical listing of all the GS BASIC error messages, complete with possible explanations for why the error occurred.

?ARGUMENT COUNT ERROR

Whenever you call a procedure with PROC, a function with FN, any external procedure with CALL, _, or PERFORM, or an external function with EXFN or EXFN_, BASIC already has recorded the number of parameters required in the referencing parameter list. This error occurs when the wrong number of parameters is used.

?ARGUMENT TYPE MISMATCH ERROR

Whenever you call a procedure with PROC, a function with FN, any external procedure with CALL, _, or PERFORM, or an external function with EXFN or EXFN_, BASIC already has recorded the type of each parameter required in the referencing parameter list. This error occurs when a parameter of the wrong type is used. Normally, numeric parameters are converted to the proper type, so this error usually refers to a string argument used for a numeric parameter or vice versa.

?BAD PATH ERROR

This error message will be displayed if you specify an illegal character in any part of a pathname.

?BAD SUBSCRIPT ERROR

An attempt was made to reference an array element that is outside the bounds of the array. There are two primary causes of this error:

- attempting to access a nonexistent dimension in an array; for example:

```
)DIM  TIMES(23,59)
)TIMES(11,30,27)=TRIGGER
```

- attempting to access a nonexistent element in any dimension; for example,

```
)DIM  DATES(11,30,2000)
)DATES(8,28,2001)=-1
```

?CAN'T CONTINUE ERROR

This error will occur if you are attempting to continue a program after modifying anything other than the variables in the current program.

?CAN'T RENUMBER ERROR

The RENUM command issues this error if insufficient memory is allocated in the user data segment to provide space for the line number cross-reference table needed to reunumber the program in memory. This error is also generated if the RENUM arguments request an impossible renumber task; that is, one that would generate line-number conflicts.

?DAMAGED DIRECTORY ERROR

This error occurs when ProDOS discovers that the number of entries indicated in the directory header does not match the number of entries the directory actually contains.

?DAMAGE REPORT ERROR

This error message is generated when the string garbage collector discovers that the string literal data pool has been damaged and it is unable to properly readjust the string descriptors in the variable tables to point at the string literal data. A program will not probably generate correct output after this error occurs. A program could Clear its data and restart to correct this rare condition.

?DEVICE NOT CONNECTED ERROR

This error occurs when an interface card for a device is present in your system, but the requested device is not physically cabled to the interface card.

?DEVICE NOT FOUND ERROR

There are two possible causes of this error:

- Your system is not configured for the device you specify.
- The device name you specify includes an illegal character.

?DIFFERENT VOLUMES ERROR

The pathnames of a RENAME specify two different volumes. RENAME can only move files amound directories on the same volume.

?DIRECTORY FULL ERROR

This error occurs when all the entries (55) in a root directory of a ProDOS disk volume are in use and an OPEN or CREATE tries to add another one.

?DIR PATTERN ERROR

This error occurs if an invalid wildcard pattern filename is used with the DIR.

?DISK FULL ERROR

There is no space left for additional information on the disk. Either delete files or use another disk volume.

?DIVISION by ZERO ERROR

The dividend of any number divided by 0 is infinity. Infinity is mathematical concept that SANE supports by providing a representation for it in single or double-precision real variables, but not in integer variables.

This error occurs when a division by zero is attempted, unless the division by zero exception is suppressed and the division is done with real operands, in which case infinity is generated in the reak result.

?DRIVE EMPTY ERROR

This error occurs due to access to a removable media disk drive device lacking a disk volume.

?DUPLICATE DEFINITION ERROR

There are numerous possible causes, the most common are:

- After an array was defined, another DIM statement for the same array was executed. This error often occurs if an array has been given the default dimension 10 because a statement such as A(I)=3 is followed later in the program by a DIM A(100).
- A program contains two DEF statements with the same function or procedure name. This occurs during RUN or CHAIN commands.
- An invokable module DICTIONARY contains an entry-point name that already exists in the invoke library dictionary table.

?DUPLICATE FILE ERROR

An attempt was made to rename a file to a name that already exists on the current disk volume.

?DUPLICATE LABEL ERROR

This error occurs when a program line is entered with the same line label already present on a line with a different line number.

?DUPLICATE VOLUME ERROR

This error occurs when ProDOS recognizes that it has two mounted volumes with identical volume names. This message is a warning; it does not prevent access to either volume. The ambiguous volume selection problem is be resolved by using the first volume found (how is first defined???).

?END FN/PROC w/o DEF ERROR

The error occurs during the DEF scan executed by RUN or CHAIN and if an END FN or END PROC statement occurs outside a procedure or function definition.

?EXTRA IGNORED

More values were supplied than were asked for by the variable list of an INPUT statement. This is actually a warning, not an error.

?FILE CREATE ERROR

This error results during an OPEN that creates a new file and then is unable to read the new directory entry immediately thereafter. This error implies that a volume directory or subdirectory was written sucessfully but was unreadable on the next read. Something major is probably wrong with the drive or its media.

?FILE LOCKED ERROR

An attempt was made to modify a locked file.

?FILE NOT FOUND ERROR

An attempt was made to access a file that does not exist on the disk. The other elements of the file's pathname must exist for this error to be generated. If a volume name, a prefix, or a subdirectory name in the file's pathname are wrong or do not exist, this error will not occur.

?FILE NOT OPEN ERROR

An attempt was made to access a file, via a file reference number, before opening it with the OPEN statement.

?FILE TOO LARGE ERROR

The LIBRARY statement generates this error when it opens a TDF file and the file is larger than 64K.

?FILE OPEN ERROR

This error will occur if you attempt to delete or rename a file while it is open.

?FILE TYPE ERROR

This occurs when the file type attribute of a disk file is incompatible with the implied purpose of the referencing statement. There are too many contexts that cause this error to list them here; refer to the definition of the offending statement to determine which file types are allowed.

?FOR w/o NEXT ERROR

This error occurs when a FOR statement is unable to locate, by scanning forward, a NEXT statement at the nesting level matching that of the FOR statement about to be executed. Refer to the FOR statement description for requirements and restrictions on how the matching NEXT statement is located.

?FORMULA TOO COMPLEX ERROR

This error has two possible causes:

- parentheses in an expression are nested more than 14 deep
- an attempt was made to evaluate an arithmetic expression with more than 14 pending operations caused by precedence

?INEXACT ERROR

This error is normally disabled. It will only occur if the SANE exception mask is changed to enable the inexact exception with the EXCEPTION ON statement.

?ILLEGAL DIRECT ERROR

Given when a DEF FN, DATA, EVENTDEF, RESUME, MENUDEF, ON BREAK, ON ERR, ON EXCEPTION, ON KBD, ON EOF#, ON TIMER, or TASKPOLL statement is used in immediate execution; these statements may only be used in program statement.

?ILLEGAL LINE NUMBER/LABEL ERROR

Some of the causes for this error are:

- The line number or label used in a statement, such a GOTO or GOSUB, contained an invalid character or was out of range (negative, zero, or more than 65279).
- The line number or label used when a statement was entered contains an invalid character, is out of range, or the label is a reserved word.
- The line number or label referenced in the USING clause of a statement is out of range or contains an invalid character.

?ILLEGAL QUANTITY ERROR

The parameter passed to a function or used with a statement was out of range. Illegal quantity errors can be caused by:

- a negative array subscript (for example, A(-1))
- using MID$, LEFT$, RIGHT$, VPOS, HPOS, SPC, WINDOW, TAB, SUB$, CHR$, HEX$, TEN, INSTR, SCALE, or ON ... GOTO with an expression whose value lies outside the allowable range
- opening a file with a record length less than 3
- specifying a file number less than 1 or greater than 10
- using a repetition value greater than 255 in a PRINT(#) USING statement or an IMAGE specification
- a value with an integer range (-32768 to 32767) was expected, but a value beyond that range was encountered

?INPUT USING PARM ERROR

INPUT USING issues this error when the wrong type of parameter is used in the IMAGE statement for a specific parameter.

?INT/FCB/VCB TBL FULL ERROR

This is really three different error messages lumped together; they all occur very rarely:

- The interrupt vector table (INT TBL FULL ERROR) message appears when one too many ProDOS Allocate Interrupt calls is made to ProDOS. ProDOS 16 supports 16 user interrupt handlers.

  The file control block (FCB TBL FULL ERROR) message appearswhen you attempt to OPEN one too many disk files. ProDOS 16 version 1.2 is limited to 8 total. BASIC allows up to 29 such files in anticipation of a future release of ProDOS that will support more than 8 open files, so BASIC does not check how many files you open. BASIC needs 1 file always left unused for the CAT, DIR, CATALOG, or TYPE command, plus 1 file if you expect to use EXEC files.

Warning

A BASIC program should not open more than six disk files at once with ProDOS 16 version 1.2.

The volume control block (VCB TBL FULL ERROR) message appears when the VCB table already contains 8 volumes/devices. The error occurs when 8 devices/volumes are online and a ProDOS VOLUME call is made for another device that has no open files. This error is very rare since most systems don't have 9 usable disk drives or harddisks partitioned into many volumes.

? = ADRS: INTERNAL ERROR

An internal cross-check of the correct function of the interpreter or its data structures has failed. GS BASIC performs certain self consistency checks for some of its internal operations. On the line above the error message is a relative address within the interpreter of where the check that failed was made. This error indicates that some important data has been corrupted, and a program should terminate if this occurs.

This error message may indicate, if the problem consistently recurs, the presence of a software fault in the BASIC interpreter, ProDOS, a tool set, or an external module that you are using.

?INVALID DATA ERROR

All of the following will generate this error:

- An invalid BDF field tag byte was found during a read operation.
- The parameters for the VAR function attempted to extract a variable from a structure too close to the end of the structure element to extract the the requested size variable.
- The DICTIONARY segment of an invokable module does not contain the proper format version number.
- A tool set dictionary interface-definition record contains invalid data for an input parameter type byte or a error handling mode byte.

?INVALID DEVICE ERROR

The disk device name used is a valid disk device name, but the device number is larger than the total number configured in your system. for example, CAT .D39.

?I/O ERROR

A physical operation of a peripheral device failed. The most common cause is the failure of a disk drive to sucessfully read or write the disk media; this usually indicates a defective disk volume.

It also could be a mechanical or electrical problem causing a loss of data; check all external device connections for possible problems (is everything plugged in properly?).

?LINE TOO LONG ERROR

There are two possible causes of this error:

- You have entered a line with so many CALL or _ verbs that the extra hidden field used to optimize CALL statement performance will cause the line to exceed the line size limit of 255 bytes.
- RENUM would have to insert so many bytes into a line to renumber it that the line would exceed 255 total bytes (including overhead).

?MENU or EVENT ERROR

This error is generated when a TaskMaster event occurs and the line defined for that menu item or event code does not exist, even though it has been defined via a MENUDEF or EVENTDEF statement.

?MISSING END PROC/FN ERROR

The DEF scan executed during a RUN or CHAIN command issues this message when it can't find the END PROC or END FN statement matching a DEF statement.

?MISSING OPERAND ERROR

The following causes generate this error:

- There are insufficient parameters in the IMAGE of an INPUT USING statement.
- A user-defined function is referenced without a parameter list enclosed in parenthses.
- EXFN is used without a function name

?MISSING RETURN ERROR

This error message is issued when an END PROC or END FN is executed and a GOSUB is still pending on the top of the control stack. A GOSUB was executed within a procedure or function, but the matching RETURN was never executed.

?MULTI-LINE FN REF ERROR

This error message occurs when a multiline function is referenced anywhere in a program other than in a LET or FN LET expression.

?MULTI STATEMENT ERROR

The DEF statement scan done during RUN and CHAIN requires that an END PROC or END FN statement, that begins a program line, has no other statements following the END statement in that program line.

?NEXT w/o FOR ERROR

There are three possible causes of this error:

- Loops are nested improper; control variables in a NEXT statement must be listed in the reverse order that they were encountered in FOR statements.
- The control variable specified in a NEXT statement does not correspond to the variable in any FOR statement still in effect.
- A NEXT statement without a specified control variable was executed when no FOR statement was in effect.

?NESTED DEF ERROR

This error occurs when the DEF scan, executed by RUN or CHAIN, finds a DEF statement within the body of procedure or function. It probably indicates that an END PROC or END FN statement is missing.

?NO LIBRARY/INVOKE ERROR

This error occurs when a CALL, _ , EXFN_, PERFORM, EXFN, or LIBFIND statement is used without first using LIBRARY or INVOKE. It indicates that the dictionary for tool sets or invokable modules is empty.

?NO SEGMENT ERROR

This error occurs when INVOKE is informed by the System Loader that the invokable module does not contain a segment named DICTIONARY or does not contain a code segment with segment #01.

?NOT A NUMBER ERROR

This error is generated when the IF statement encounters a numeric comparison operand that is a NaN (Not A Number). See Appendix K, "SANE Considerations," for details.

?NOT LOCAL ERROR

This error occurs when a variable reference is global that must be a local variable. The most common case is attempting to use global variable as the target of a FN = assignment statement.

?OUT of DATA ERROR

There are three possible causes of this error.

- A READ statement was executed, but all the data elements in DATA statements in the program have already been read.
- A READ# or INPUT# statement ran out of data when reading from a file; in other words, an end of file was reached.
- A record or field being sent to a file is longer than the one specified for the file.

?OUT of MEMORY ERROR

There are numerous possible causes of this error, the most common are:

- There is no memory available for a file buffer when you open a file.
- The program you tried to LOAD, RUN, or CHAIN is too large.
- The LIBRARY statement cannot obtain memory to load a TDF.
- An invoked file will not fit in the available memory space.
- Arrays or variables called for need more space than is allocated to the user data segment, see the CLEAR statement definition.

?OVERFLOW ERROR

The result of a calculation was too large to be represented in a specific numeric format. This occurs when attempting to store a number too large for a specific type of integer or real variable.

?PATH NOT FOUND ERROR

Part of the pathname specified did not exist it the directory structure of the referenced volume. It normally occurs because of a misspelled subdirectory name.

?POSITION RANGE ERROR

A disk file I/O statement has attempted to reference a record in a file that is beyond the current end-of-file mark for a disk file. Check the value of the record number and verify that the correct file number was used. This error is most often associated with the READ #, WRITE # , GET #, or PUT # statements.

?PROC NAME ERROR

This error occurs when a procedure name is followed by a type character. Procedure names can not have types like function names.

?ProDOS CALL ERROR =$xx

An error occurred within the operating system, ProDOS, that is not assigned a specific message by BASIC. The ProDOS hexadecimal error code is shown following the equal sign. The exact error message can be found in the *ProDOS 16 Reference* manual.

Generally, this error indicates that something that should not normally happen has occurred. If the error can be repeated every time a specific sequence of events or inputs is used, even after powering down and up again, it may indicate the presence of a software fault in the interpreter, the operating system, a tool set, an invokable module, or a BASIC program that is poking around in the wrong places in memory.

It may also mean that the error is so uncommon and infrequent that no error message was included in BASIC for it. Refer to the *ProDOS 16 Reference* manual for descriptions of these errors.

?ProDOS VERSION ERROR

GS BASIC will only execute the CAT, CATALOG, and DIR commands when operating within the environment of the proper version of ProDOS 16, specifically version 1.2 or later.

?RANGE ERROR

There are three causes for this message:

- An illegal line range was specified in a DEL or LIST statement.
- Text output to the console could not be formatted correctly because the INDENT position was beyond the wrap column set by OUTREC.
- The SET statement issues this error when the length parameter specifies a value other than 4, 8, or 10 and an extended precision expression conversion was required.

?RECURSION ERROR

This error message is generated if the pause procedure called by the line number option of the COPY command itself executes a COPY command.

?REENTER

The characters entered in response to an INPUT request for a numeric variable are not a valid representation of a number.

?RESERVED WORD ERROR

This error occurs when a line is added to a program and any of its statements begins with a verb that may not begin a statement or where a verb within a statement may only be used to begin a statement. See the section in Chapter 1, "Syntax Checking." This error might also occur if a program image has become corrupted and certain invalid verb tokens occur as the first verb in a statement.

?RETURN 0 ERROR

This error occurs if a RETURN 0 statement is used incorrectly or without being invoked by a TaskMaster event dispatch or an external event dispatch.

?RETURN w/o GOSUB ERROR

More RETURN statements and/or POP statements were executed than GOSUB statements.

?SANE INVALID ERROR

There are numerous possible causes of this error:

- using the LOG function with a negative or zero argument
- using the SQR function with a negative argument
- using X MOD Y or X REMDR Y where Y is zero or X is infinity
- meaningless division of 0/0 or INF/INF
- meaningless multiplication of 0*INF
- magnitude subtraction of infinites, that is, (+INF)+(-INF)

For more information on this error message, see Appendix K, "SANE Considerations."

?STACK OVERFLOW ERROR

There are a number of possible causes of this error:

- FOR ... NEXT loops nested more than 9 deep
- GOSUB subroutines, DO ... UNTIL, WHILE ... UNTIL, DO ... WHILE ... UNTIL loops, PROCs or multiline function calls nested more than 40 deep

- ON KBD subroutines entered more than 40 times without a RETURN
- ON TIMER subroutines entered more than 40 times without a RETURN

?STRING TOO LONG ERROR

The value of a string expression is greater than 255 characters in length.

?SYNTAX ERROR

Any of the following can cause this error:

- missing parenthesis in an expression
- illegal character in a statement
- ON not followed by GOTO or GOSUB
- IF not followed by THEN or GOTO
- arithmetic operation on a string
- a digit as the first character of variable name
- variable name more than 29 characters in length
- bad specification in an IMAGE format
- bad FOR option for OPEN
- bad operator
- following DEL with something other than a digit
- a valid reserved word used in the wrong context or order
- anything else that is not syntactically correct

?STRING SPACE ERROR

This error occurs if a string parameter address setup on the stack to be passed by reference to an external subroutine had been invalidated because of garbage collection caused by subsequent string expression evaluation in another string parameter. The only solution to this problem is not to use string expressions as arguments.

?TASKPOLL INIT &/ WINDSTARTUP ERROR

This error occurs when TASKPOLL ON or TASKPOLL INIT are executed and the prerequisite Window Manager environment has not been set up first.

?TOOLSET CALL ERROR =$TTEE

This error results when an external tool set procedure or function call, made via CALL, _, or EXFN_, returns an error. The error number is shown following the message in hexadecimal. The TT portion of the error number is the tool set number, and the EE portion is the actual tool set specific error. Refer to the *Apple IIGS Toolbox Reference* manual for details.

?TYPE MISMATCH ERROR

Any of the following can cause this error:

- The left side of an assignment statement was a numeric variable and the right side was a string, or vice versa.
- A function that expected a string argument was given a numeric one, or vice versa.
- The wrong IMAGE specification for string/numeric was used in PRINT(#) USING statement.
- A READ# numeric data instruction was encountered when next data are a string, or vice versa.

?UNCLAIMED EVENT ERROR

There are two causes for this error:

- An ON KBD, ON BREAK, ON TIMER, or ON EXCEPTION is called to dispatch an event, and no event-handler line number has been defined for the event. This indicates that a control flag has been enabled for an event, without the matching line number and pointer having been set up also.

?UNDERFLOW ERROR

This error occurs when the result of a SANE math computation is too small to be represented. This is a rare occurrence because SANE supports a very large range of numbers.

?UNDEFINED ARRAY ERROR

A number of different causes generate this error:

- Referencing a structure array in a SET, GET#, or PUT# statement before it is dimensioned will generate this error. Unlike numeric arrays, Structure arrays must be dimensioned before they are used.
- The UBOUND function was called with an array that has never been dimensioned.
- Attempting to ERASE an array that has never been dimensioned will generate this error.

?UNDEF'D PROC/FUNCTION ERROR

This error can occur for the following reasons:

- Reference was made to a user-defined function that had never been defined or before the program was ever been run.
- Reference was made (with the PROC statement) to a user defined procedure that doesn't exist or has been entered since the last time the program was run.
- Reference was made with CALL, _, or EXFN_ to an external procedure or function in a tool set without first loading the interface definition(s) and/or the tool set with the LIBRARY statement.
- Reference was made with PERFORM or EXFN to an external assembly-language procedure or function in a module without having loaded the interface definition(s) and the module with the INVOKE statement.

?UNDEF'D STATEMENT ERROR

Any of the following can cause this error:

- An attempt was made to GOTO, GOSUB, or THEN to a statement line number or label that does not exist or has been deleted.
- PRINT USING line was used when the line does not exist.
- IMAGE is not the first statement in the line, or the IMAGE list is null.

?UNTIL w/o WHILE ERROR

There are two possible causes:

- Conditional WHILE ... UNTIL, DO ... UNTIL or DO ... WHILE ... UNTIL loops were improperly nested.
- An UNTIL statement was executed when no DO ... WHILE loop was in effect.

?VARIABLE ERROR

A number of different causes generate this error:

- A string array has been allocated in the array table beyond the first 64K of array memory. String arrays should be allocated first and must all fit within the first 64K. The error occurs when an assignment to the array is attempted.
- A string variable has been allocated in the simple variable table beyond the first 64K of array memory. String variables should be allocated first and must all fit within the first 64K of the simple variable table. The error occurs when an assignment to the variable is attempted.
- Attempting to use a FOR loop control variable that is allocated beyond the first 64K of the simple variable table origin will cause this error.

- An ordinary variable was used in a statement in place of a required structure array reference (with a subscript) such as SET, GET#, and PUT#.
- Attempting to ERASE a variable that has never been defined will cause this error.
- An array name was used in a LOCAL statement (where only a simple variable is allowed).
- A null string parameter was passed to the VARPTR$ function.

?VOLUME NOT FOUND ERROR

The volume name specified in the pathname of an I/O statement does not match the volume name of any currently mounted disk volume.

?VOLUME SWITCHED ERROR

The disk volume for an I/O operation (on an open file) has been removed from its drive; the operation cannot be completed.

Warning

Unlike Apple /// drives, most Apple II drives have no hardware to detect disk switches. This error is therefore returned only when ProDOS checks a volume name during the normal course of an I/O call. Because most disk I/O calls do not involve a volume name check, many disk-switched errors go undetected.

?VOLUME TYPE ERROR

The disk volume being accessed is readable but is not formatted as a ProDOS (or SOS) disk volume. This usually implies that the volume is formatted for DOS 3.3 or Apple II Pascal.

?WHILE w/o UNTIL ERROR

This error occurs when a WHILE statement is unable to locate, by scanning forward, an UNTIL statement at the nesting level matching that of the WHILE statement about to be executed. Refer to the WHILE statement description for requirements and restrictions on how the matching UNTIL statement is located.

?WRITE PROTECT ERROR

The files on the disk volume cannot be modified because it is write-protected.

ERROR CODES

When BASIC detects an error, the reserved variable ERR will contain a number code corresponding to the following table.

Table B-1
Error codes and messages

Error Code	Message
1	NEXT w/o FOR
2	SYNTAX
3	RETURN w/o GOSUB
4	OUT of DATA
5	ILLEGAL QUANTITY
6	INVALID DATA
7	ILLEGAL LINE NUMBER/LABEL
8	DUPLICATE LABEL
9	OVERFLOW
10	OUT of MEMORY
11	UNDEF'D STATEMENT
12	BAD SUBSCRIPT
13	RANGE
14	STACK OVERFLOW
15	DUPLICATE DEFINITION
16	DIVISION by ZERO
17	ILLEGAL DIRECT
18	TYPE MISMATCH
19	STRING TOO LONG
20	FORMULA TOO COMPLEX
21	CAN'T CONTINUE
22	UNDEF'D PROC/FUNCTION
23	VARIABLE
24	TOOLSET CALL ERROR = $ttee
25	ProDOS CALL ERROR = $ee
26	FILE OPEN
27	VOLUME TYPE
28	DRIVE EMPTY
29	FILE TYPE
30	I/O
31	FILE TOO LARGE
32	WRITE PROTECT
33	VOLUME SWITCHED
34	BAD PATH
35	FILE NOT FOUND

36	PATH NOT FOUND
37	VOLUME NOT FOUND
38	DUPLICATE FILE
39	DISK FULL
40	FILE LOCKED
41	FILE NOT OPEN
42	DEVICE NOT CONNTECTED
43	INT/FCB/VCB TBL FULL
44	DIRECTORY FULL
45	DUPLICATE VOLUME
46	= ADRS: INTERNAL ERROR
47	FOR w/o NEXT
48	POSITION RANGE
49	FILE CREATE
50	DIFFERENT VOLUMES
51	DAMAGED DIRECTORY
52	LINE TOO LONG
53	RESERVED WORD
54	ARGUMENT COUNT
55	ARGUMENT TYPE MISMATCH
56	UNDEFINED ARRAY
57	WHILE w/o UNTIL
58	UNTIL w/o WHILE
59	MULTI STATEMENT
60	MISSING OPERAND
61	NESTED DEF
62	RECURSION
63	MISSING END PROC/FN
64	END PROC/FN w/o DEF
65	MISSING RETURN
66	DAMAGE REPORT
67	NOT LOCAL
68	MULTI-LINE FN REF
69	PROC NAME
70	INPUT USING PARM
71	UNCLAIMED EVENT
72	SANE INVALID
73	INEXACT
74	UNDERFLOW
75	NOT a NUMBER
76	NO SEGMENT
77	NO LIBRARY/INVOKE
78	STRING SPACE
79	MENU or EVENT
80	TASKPOLL INIT &/ WINDSTARTUP

81	DEVICE NOT FOUND
82	INVALID DEVICE
83	ProDOS VERSION
84	RETURN 0
85	DIR PATTERN
86	CAN'T RENUMBER
87	Not implemented yet
253	EXTRA IGNORED
254	REENTER

Appendix C

Reserved Words

The following is an alphabetical list of the reserved words in Apple IIGS BASIC. Note that some must end with a left parentheses to be considered reserved words. For example, AND is an illegal variable name, but ABS is not.

_ (underscore)
ABS(
AND
ANU(
APPEND
AS
ASC(
ASSIGN
ATN(
AUTO
AUXID@
BASIC@(
BDF
BREAK
BTN(
CALL
CALL%
CAT
CATALOG
CHAIN
CHR$(
CLEAR
CLOSE
COMPI(
CONT
CONV(
CONV@(
CONV#(
CONV$(
CONV%(
CONV&(
COPY
COS(
CREATE
DATA
DATE$
DATE(
DEF
DEL
DELETE
DIM
DIR
DIV
DO
EDIT
ELSE
END
EOF
EOFMARK(
ERASE
ERR
ERRLIN
ERROR
ERRTOOL
ERRTXT$(
EVENTDEF
EXCEPTION
EXEC
EXEVENT@(
EXFN
EXP(
EXP1(
EXP2(
FILE(
FILTYP(
FILTYP=
FIX(
FN
FOR
FRE
FREMEM(
GET
GOSUB
GOTO
GRAF
HEX$(
HLIST
HOME
HPOS
IF
IMAGE
INDENT
INIT
INPUT
INSTR(
INT(
INVERSE
INVOKE
JOYX(
JOYY
KBD
LEFT$(
LEN(
LET
LIBFIND
LIBRARY
LIST
LISTTAB
LOAD
LOCAL
LOCATE
LOCK
LOG(
LOG1(
LOG2(
LOGB%(
MENUDEF
MID$(
MOD
NEGATE(
NEW
NEXT
NORMAL
NOT
NOTRACE
OFF
ON
OPEN
OR
OUTPUT
OUTREC
PDL(
PDL9
PEEK(
PERFORM
PFX$(
PI
POKE
POP
PREFIX
PREFIX$
PRINT
PROC
PROGNAM$
PUT
QUIT
R.STACK%(
R.STACK@(
R.STACK&(
RANDOMIZE
READ
REC(
RELATION(
REM
REMDR
RENAME
REP$(

RESTORE
RESUME
RETURN
RIGHT$(
RND(
ROUND(
RUN
SAVE
SCALB(
SCALE(
SECONDS@
SET
SGN(
SHOWDIGITS
SIN(
SPACE$(
SPC(
SQR(
SRC
STEP
STOP
STR$(
SUB$(
SWAP
TAB(
TAN(
TASKPOLL
TASKREC%(
TASKREC@(
TEN(
TEXT
TEXTPORT
THEN
TIME$
TIME(
TIMER
TO
TRACE
TXT
TYPE
TYP(
UBOUND(
UCASE$(
UIR(
UNLOCK
UNTIL
UPDATE
USING
VAL(
VAR(
VAR$(
VARPTR(
VARPTR$(
VOLUMES
VPOS
WHILE
WRITE
XOR

VERBS

A verb is a reserved word that must begin a statement as the statement verb or as a modifiable reserved variable. Those words shown in **bold** are verbs also used as adverbs to begin a clause following some other verb.

Words marked with an asterisk are only commands; that is, they may **not** occur in statements (except as an adverb).

_ (underscore)
ASSIGN
AUTO*
BREAK
CALL
CALL%
CAT
CATALOG
CHAIN
CLEAR
CLOSE
CONT*
COPY
CREATE
DATA
DATE$
DEF
DEL*
DELETE
DIM
DIR
DO
EDIT*
ELSE
END
ERASE
ERROR
EVENTDEF
EXCEPTION
EXEC
FN
FOR
GET
GOSUB
GOTO
GRAF
HLIST*
HOME
HPOS
IF
IMAGE
INDENT
INIT
INPUT
INVERSE
INVOKE
LET
LIBFIND
LIBRARY
LIST*
LISTTAB
LOAD
LOCAL
LOCATE
LOCK
MENUDEF
NEW
NEXT
NORMAL
NOTRACE
OFF
ON
OPEN
OUTPUT
OUTREC
PERFORM
POKE
POP
PREFIX
PREFIX$
PRINT
PROC
PROGNAM$
PUT
QUIT
RANDOMIZE
READ
REM
RENAME
RESTORE
RESUME
RETURN
RUN
SAVE
SET
SHOWDIGITS
STOP
SUB$(
SWAP
TASKPOLL
TEXT
TEXTPORT
THEN
TIME$
TIMER
TRACE
TYPE
UNLOCK
UNTIL
VOLUMES
VPOS
WHILE
WRITE

Adverbs

The following is an alphabetical list of the adverbs used at the beginning of a clause in the syntax of a statement begun with a verb. Those adverbs shown in bold are also used a a verb to begin a statement.

APPEND
AS
AUTO
BREAK
CAT
COPY
EOF
ERR
ELSE
EOF
ERR
EXCEPTION
FILTYP=
FN
FOR
GOTO
GOSUB
INIT
INPUT
INVOKE
KBD
LIBRARY
NEXT
OFF
ON
OUTPUT
PROC
STEP
TEXT
THEN
TIMER
TO
UPDATE
USING

Operators

The following is an alphabetical list of the operators used in arithmetic or logical expressions.

AND
DIV
MOD
REMDR
NOT
OR
XOR

Nouns

The following is an alphabetical list of the predefined nouns of IIGS Basic. A noun is a reserved word that can be used in an expression or variable list of a statement (following a verb). All nouns are characterized by having or imply either a numeric or string value as is appropriate.

A few nouns in the list shown in **bold** are also verbs and reserved variables.

ABS(
ANU(
ASC(
ATN(
AUXID@
BASIC@(
BDF
BTN(
CHR$(
COMPI(
CONV(
CONV@(
CONV#(
CONV$(
CONV%(
CONV&(
COS(
DATE$
DATE(
DIR
EOF
EOFMARK(
ERR
ERRLIN
ERRTOOL
ERRTXT$(
EXFN
EXP(
EXP1(
EXP2(
FILE(
FILTYP(
FIX(
FN
FRE
FREMEM(
HEX$(
HPOS
INDENT
INSTR(
INT(
JOYX(
JOYY
KBD
LEFT$(
LEN(
LISTTAB
LOG(
LOG1(
LOG2(
LOGB%(
MID$(
NEGATE(
OUTREC
PDL(
PDL9
PEEK(
PFX$(
PI
PREFIX$
PROGNAM$
R.STACK%(
R.STACK@(
R.STACK&(
REC(
RELATION(
REP$
RIGHT$(
RND(
ROUND(
SCALB(
SCALE(
SECONDS@
SGN(
SIN(
SPACE$(
SPC(
SQR(
SRC
STR$(
TAB(
TAN(
TASKREC%(
TASKREC@(
TEN(
TIME$
TIME(
TXT
TYP(
UBOUND(
UCASE$(
UIR(
VAL(
VAR(
VAR$(
VARPTR(
VARPTR$(
VPOS

Appendix D

INTERPRETER DATA STRUCTURES

Memory Usage in Variable Tables

BASIC stores the values for variables and arrays in three partitions within the user data segment. Arrays are stored separately from simple and local variables. The entries in these tables are described here in detail primarily for programmers who may want to process the tables directly. This discussion also specifieshow much space each variable or array will require.

Simple variable format

Every simple variable in Apple IIGS BASIC has an entry in the simple variable partition of the user data segment. Local variables are also stored in their own partition; both types are stored with the following format:

| LENGTH | NAME | TYPE | VALUE |

LENGTH is a 1-byte field that contains the size of the entire variable entry in bytes.

NAME is a field of variable length that contains the ASCII code of the simple variable name. NAME is between 1 and 30 bytes in length.

TYPE is a 1-byte field that contains a code for the type of the variable.

Table D-1

TYPE value	Variable type	Value size
$C1	single integers	2 bytes
$C2	double integers	4 bytes
$C3	long integers	8 bytes
$84	double precision real	4 bytes
$85	single precision real	8 bytes
$87	strings	3 bytes

VALUE is a field that contains the value of the variable. The length and contents of the VALUE field depend on the variable type, as shown. All data values are stored with the least significant byte in lowest memory.

The value field for a string consists of a 1-byte string length and a 2-byte relative offset to the origin of the literal string data in the literal pool. The base for this relative offset is the end of the literal pool (highest address + 1).

Array Variable Format

Every array variable in BASIC has an entry in memory with the following format:

| LENGTH | NAME | TYPE | S.COUNT | D.SIZE | VALUES |

LENGTH is a 3 byte field that contains the size of the entire array variable entry in bytes.

NAME is a field of variable length that contains the ASCII code of the array variable name. NAME is between 1 and 30 bytes in length.

TYPE is a 1-byte field that contains a code for the type of the simple variable. The same TYPE codes as shown for simple variables are used, with the addition of one value, $C0 for Structure arrays (byte arrays).

S.COUNT is a 1-byte field that contains the number of subscripts in the array variable.

D.SIZE is a field that contains the size of each dimension in the array. Its length, in bytes, is equal to the number of dimensions times two.

VALUES is a field containing the values of each of the array elements. The array elements are stored with the rightmost index ascending slowest. Each element in the array occupies the number of bytes shown for the value fields of simple variables, in the section above. Structure arrays (TYPE = $C0) are 1 byte per element. TYPE = $87 strings are 3 bytes per element.

Memory usage in programs

This section discusses how much memory space is used by GS BASIC to store variables and constants. Note that the information given here is not essential to learning how to program.

A byte is the smallest individually accessible unit of memory, and each byte contains eight binary digits, or bits. A 512K Apple IIGS has 524,288 bytes of memory. After BASIC, ProDOS 16, the video buffers, and work areas for the tools have been allocated space, there is about 192K of memory available for you to use.

Constants

Integer constants with one to nine digits are converted into binary when a program line is entered (and converted back into characters for listing). An integer is converted to binary so that the binary form is always no larger than the characters you enter. An integer is tokenized only if it is a contiguous string of one to nine digits.

Integers with one digit are stored as 1 byte; integers with values from 10 through 4095 are stored as 2 bytes; integers with values from 4096 through 65535 are stored as 3 bytes; and values from 65536 through 999999999 are stored as 5 bytes. If an integer is preceded by a plus or minus sign, that character occupies 1 additional byte.

If the number has more than nine digits, it occupies 1 byte of memory in a program for each digit.

All real constants require 1 byte of memory per digit, including the characters ., +, -, and E. For example, the constant

2.7182818E+46

uses 13 bytes.

Numeric variables

One byte of memory is used for each character of a variable name (up to a maximum of 30 bytes), plus an additional byte is used if the variable name has a type character.

The memory requirements for a variable in the user data segment are described earlier, in the "Memory Usage in Variable Tables," section.

Strings

BASIC stores strings in two parts. The first part is the entry in the variable tables, and the second part contains the actual sequence of characters in the string. The entry in the variable table includes a 3-byte string descriptor, as described earlier in the section, "Memory Usage in Variable Tables."

In addition to the 3 bytes for the descriptor, each string variable uses 1 byte for each character of the string name (except the final dollar sign), 1 link byte, and 1 type byte, plus 1 byte for each character in the string. An additional 3 bytes are needed for overhead, unless the value of the variable is the null string. For example, the statement

```
)TR$ = ""
```

uses 7 bytes: 3 for the string descriptor, 2 for the string name, 1 for the link, and 1 for the variable type. The statement

```
)STAR$ = "WIX"
```

uses 15 bytes: 3 for the string descriptor, 4 for the string name, 1 for the link, 1 for the variable type character, 3 for the string characters, plus 3 overhead bytes.

IIGS BASIC reserves part of memory solely for strings. Exactly how much memory is available for string storage depends on how many other variables and arrays exist in a program and the size of the program itself. When a string decreases in length, BASIC does not immediately reclaim the freed memory space. Instead, whenever BASIC sees that it is about to run out of user data segment memory, it compacts string storage to recover space previously abandoned. All the recovered string storage space is then reused by new strings.

You can cause BASIC to reorganize memory space by referencing the reserved variable FRE.

Arrays

Each array requires the following amount of memory: 3 link bytes, 1 type byte, 1 byte per character of the array name, 1 byte recording the number of array dimensions, and 2 bytes per dimension.

Each integer array element occupies 2 bytes of memory; each double-integer array element occupies 4 bytes of memory; single- and double-real array elements occupy 4 and 8 bytes, respectively; long integer array elements occupy 8 bytes; and string array elements occupy 3 bytes (the string descriptors).

For example, the statement

```
)DIM  Paymt%(9,3,5)
```

allocates 16 bytes plus the space for the values: 3 link bytes, 1 type byte, 5 bytes for the array name, 1 byte for the number of dimensions, and 6 bytes for the three dimensions. The values occupy 10 x 4 x 6 x 2 bytes per integer element, or 480 bytes.

Program tokenization

Reserved verbs are tokenized into single bytes called tokens. All adverbs, operators, and nouns (functions and reserved variables) are converted into 2-byte tokens. Appendix C, "Reserved Words," specifies which words are verbs, nouns, adverbs, operators, and so on.

All other characters in programs each use 1 byte of storage within the program.

Each program line has 5 bytes of overhead plus the bytes described above. This overhead consists of 1 byte for the length of the optional label (plus 1 byte per label character) and 1 byte for the length of the rest of the line, including 2 bytes for the line number and 1 byte for the end-of-line token (a binary zero).

Interpreter internals

When a program is being executed, BASIC uses the space on the 256 byte FOR/GOSUB stack as follows:

- Each active FOR ... NEXT loop uses 25 bytes.
- Each active WHILE ... UNTIL loop uses 6 bytes.
- Each active GOSUB (one that has not returned) uses 6 bytes.
- Each active PROC (that has not ended) uses 6 bytes.
- Each active user defined function (that has not ended) uses 6 bytes.

The BASIC@ function

The BASIC@ function accepts an integer parameter in the range of 0 through 48. The function returns an address of some data structure in the static data areas within the GS BASIC interpreter.

The BASIC@ function allows programmers who want to work with the internal data tables to obtain the addresses of specific data structure no matter how the interpreter is implemented in the future or where the System Loader happens to place the interpreter in memory.

❖ *Note:* Do not write programs that depend on the fact that the static data segment is combined with the static code segment and will therefore reside in the same bank as the code segment. A future version of GS BASIC will probably separate the static code and data segments, and there may be more than one code segment in separate banks.

Warning

Use of any other mecanism to obtain internal addresses will not be supported in future versions, and future versions will undoubtedly return different addresses for any specific data structure defined in BASIC@.

If you are going to use POKE to manipulate the internal data structures, you should always fetch the address via the BASIC@ function. You should not capture addresses from BASIC@ into variables because future BASIC@ data structures may be dynamically relocated during interpreter execution.

BASIC@ parameter definition

The BASIC@ function returns the address of some data item for a given parameter. The assembler directive notation and data item sizes are as follows:

P-string is a count byte followed by 1 to 255 characters
DS *nnn* means define storage of *nnn* bytes; that is a buffer *nnn* bytes long
DW means define word (2 bytes, low byte first)
DB means define byte
DL means define long (4 bytes, low byte first)
rr(*directive*) means the directive inside parentheses is repeated *rr* times

Table D-2
Tabletitle

Parameter	Item Size	Item Description
0	P-string	Name of designer/developer
1	P-string	Names of major contributors
2	P-string	Names of prior authors
3	DS 128	NAMBUF: Work buffer with filenames as p-strings
4	DS 60	Startup filename GSB.HELLO
5	DS 256	MEMBUFR: Pseudo-device memory buffer
6	DS 32	LINLABL: Buffer with a statement label (see LINLBLGET)
7	DS 256	BUF: Command line buffer/ tokenization buffer
8	DS 256	IBUF: Readline buffer for INPUT, INPUT#, and so on.
9	DW	COLDSTACK: Value of IIGS BASIC Top of stack
10	DS 10	XACC: The expression evaluator X accumulator

11	DS 10	YACC: The expression evaluator Y accumulator
12	DB	OUTREC: The width of the output device for LIST command.
13	DB	INDENT: The FOR ... NEXT indentation amount
14	DB	LISTTAB: The left margin indent for the LIST command
15	DB	SHOWDIGITS: The PRINT statement significance control
16	DW	SMARGIN: String space margin in pages (default = 16)
17	DS 4	RNDSEED: The RND function startup seed
18	DS 10	RND function last value as a SANE extended-real value
19	DL	ON ERR deferred mode vector: Points at dispatcher
20	DL	ON ERR immediate mode vector: Points at ERRMSGDO-1
21	DW	Smallest precision compare switch (default = $00=off)
22	DW	EXCPCTRL switch (default=$8000): Enables exceptions
23	JML	SANEHALTV: Assembler-level JMP vector for SANE halts
24	DS 1024	CLHBUF: Command line history buffer
25	DL	Contains a handle to the 10 preallocated zero pages (see item 48 for address of BASIC's zero page)
26	DL	MINPAGES: Minimum size of user data segment in pages
27	DW DS 32	CALL_ERROR: Used by CALL and CALL%, followed by R.STACK return stack
28	DS 8	QUIT command ProDOS PQUIT: Call parameter list
29	DW DS 22	TM_EVTMSK: Event mask used for by TASKPOLL TM_RECORD: TaskMaster record used by TASKPOLL
30	DB	ATTNKEY: Byte containing the ASCII code of the Break key
31	DB	ATTNMODFR: Byte containing the modifier for Break key values: 0 = none, 1 = Shift, $10 = Keypad, $11 = Shift+Keypad
32	DL	ON TIMER: Countdown counter
33	DL	SECONDS@: Seconds-since-midnight counter
34	DW	AUXID@: AUXTYP field from prior GetFileInfo in IIGS BASIC
35	10(DS 16)	CF_NAMES: Table of 10 character device names (each entry is a P-string with 1 to 15 characters)
36	10(DB,DB)	CF_SLOT: Table of 10 device slot numbers/AUTO-LF
37	7(DW)	CF_IHOOK: Table of last input hook for slots 1 through 7
38	7(DW)	CF_OHOOK: Table of last output hook for slots 1 through 7
39	JMP long	ONJVECTOR: JMP LONG in mid ON verb processing logic

40	DL	IMMEDVCTR: Assembler level UIR immediate mode vector
41	DS 66	EDITICB: EDIT command UIR input control block
42	DS 66	RDLINICB: INPUT/command line UIR input control block
43	-	SWCHGOTBL: Address table of entry points for invokable modules
44	-	ASCFTYP: Table of 3-byte file type descriptors used by CAT
45	-	FTYP2ASC: Parallel table of 1-byte file types used by CAT
46	-	MTHSTRS: Table of twelve 4-byte month string used by CAT
47	-	DAYSTRS: Table of seven 4-byte day-of-week strings used by CAT
48	-	The address of IIGS BASIC'S zero page

The Memory Manager USERID for IIGS BASIC can be obtained from the GS BASIC zero-page as follows:

```
2990   USERID%=VAR(BASIC@(48)+7,2)
```

If you want to allocate Memory Manager segments in your programs, you should create subtype user ID's by adding $100, $200,$300 ... to the IIGS BASIC USERID. IIGS BASIC uses subtypes $F through $A for its internal memory segments and invokable modules.

The 10 tool set zero pages preallocated by IIGS BASIC are assigned as follows:

Table D-3
Tabletitle

Zero-page	Toolset Name
+$0,$100,$200	QuickDraw II
+$300	Event Manager
+$400	Sound Manager
+$500	Control Manager
+$600	LineEdit Manager
+$700	Menu Manager
+$800	Standard File Operation
+$900	Other

Version 1 of IIGS BASIC does not activate the tool sets or assign the zero pages shown in parenthesis, but a future version may. All GS BASIC application programs should observe these assignments when activating these tool sets.

Appendix E

Tips and Techniques

Time savers

The time savers hints listed below can improve the execution speed of your IIgs BASIC programs. Note that some of these hints are the same as the hints given in the second section of this appendix for decreasing the memory space used by your programs. This means that sometimes you can increase the speed of your programs while you improve the efficiency of their memory use.

1. Use integer control variables for any FOR loop with integral step values. A FOR I%= loop will execute the NEXT statement about six times faster than a FOR I=, FOR I#=, or FOR I&= loop. You could use a double-integer control variable, but a FOR I@= loop will only execute the NEXT statement about three times faster than the other forms.

2. Use integer constants instead of variables. Integer constants are converted into binary when a program statement is entered in a program. It takes less time to use an in-line binary constant than it does to look up a variable of any type in the variable tables. This is especially important within FOR ... NEXT loops or other code that is executed repeatedly.

❖ *Note:* Real constants, and integer constants with more than nine digits, are not preconverted to binary. Thus, it is better to use real and long-integer variables than to repeatedly convert them from text to binary.

3. Variables which are encountered first during the execution of a BASIC program are allocated at the start of the variable table. For example, this means that a statement such as

```
5  A=0 :  B=0 :  C=0
```

will place A first, B second, and C third in the variable table (assuming line 5 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable A, it will search only one entry in the variable table to find A, two entries to find B and three entries to find C.

4. Use NEXT statements without a specified control variable. NEXT is slightly faster than NEXT A because BASIC does not have to check to see if the control variable is the same as the one in the most recently executed FOR statement.

5. When BASIC branches to a new line number, one of two things happens depending on whether the line number is lower or higher than the currently executing line. If it is a lower number, such as

```
)1001 GOTO 1000
```

BASIC scans the entire program, starting at the lowest line until it finds the referenced line number (1000, in this example). If the new line number is greater than the current line number, BASIC only has to search forward from the current line. Here's an example:

```
)1001 GOTO 2048
```

6. You can make your program run faster if you use PROC statements instead of GOSUB statements. The benefit of using PROC statements results from the for DEF PROC and DEF FN scans performed when a program is run (or chained).

The DEF scan builds a dictionary of procedure and function names, and the PROC statement searches this table instead of the entire program to execute a procedure. The PROC statements can be anywhere in the program. The table is created in line-number order, so the PROC statement with the lowest line number will be found the fastest.

Space savers

The following hints can help you fit.

1. Use integer instead of real arrays wherever possible.

2. Use real variables instead of real constants (This rule does not apply to small integer constants in the range of 0 through 4095.) For example, suppose that you use the real constant 2.71828 ten times in your program. If you insert the statement

```
)10 E=2.21828
```

in the program, and use E instead of 2.71828 each time, you will save 40 bytes. This will also improve. In-line integer constants are always faster than integer variables but will not save space unless one or two-letter integer variable names are used.

3. The END statement is strictly optional. You can save 1 or 2 byte by omitting it from your programs. However, don't forget to use END to prevent your program from crashing into its own procedures or subroutines.

4. Reuse the same variables. If you have a variable T to hold a temporary result in one part of the program, and you need a temporary variable later in your program, use T again. Or, if you are asking the computer user to give a yes or no answer at two different times during the execution of the program, use the same temporary variable to store the reply.
5. Use PROC or GOSUB statements to execute groups of program statements that perform identical actions.
6. Use the zero elements of arrays, for example, A(0), BX!(0,X).
7. Use multiple statements per line. There is a small amount of overhead (5 bytes) associated with each line in the program.
8. In serious cases only, remove all the REM statement lines from your program. Remember that this will make your program harder for someone else to read, harder to debug, and generally less desirable—but you will save memory! If you do this, you should consider keeping a copy of the program with the REM statements.

See Appendix D, "Interpreter Data Structures," for an explanation of memory usage by variables and constants.

Appendix F

Useful Calculations

This appendix lists some useful mathematical functions that can be calculated easily in IIGS BASIC.

Table F-1

Function	BASIC equivalent
Secant	`DEF FN SEC#(X#) = 1/COS(X#)`
Cosecant	`DEF FN COSEC#(X#) = 1/SIN(X#)`
Csotangent	`DEF FN COTAN#(X#) = 1/TAN(X#)`

Inverse Sine	`DEF FN ARCSIN#(X#) : LOCAL Y#,C#`
	`C# = SCALB(-33,1.0#)`
	`Y# = ABS(X#)`
	`IF Y# >= .5#`
	`THEN Y# = ATN(X# / SQR(2*(1-Y#)-(1-Y#^2))`
	`ELSE IF Y >= C`
	`THEN Y# = ATN(X#/(SQR(1-Y#^2))`
	`ELSE Y# = X#`
	`FN ARCSIN# = Y#`
	`END FN ARCSIN#`

Inverse Cosine	`DEF FN ARCCOS#(X#) = 2 * ATN(SQR((1-X#)/(1+X#)))`
Invers Cotangent	`DEF FN ARCCOT#(X#) = ATN(X#)+1.5708`
Hyprebolic Sine	`DEF FN SINH#(X#) = (EXP(X#)-EXP(-X))/2#`
Hyprebolic Cosine	`DEF FN COSH#(X#) = (EXP(X#) + EXP(-X))/2#`

Hyprebolic Tangent

```
DEF FN TANH#(X#) : LOCAL Y#,C#
C# = SCALB(-33,1)
Y# = ABS(X#)
IF Y# > C# THEN Y#=EXP1(-2 * Y#) : Y# = -Y#/(2+Y#)
FN TANH# = Y#*SGN(X#)
END FN TANH#
```

Appendix G

Summary of IIIIGS BASIC

Syntax notation

The method used in this Appendix to describe IIGS BASIC syntax is a simple language in itself, called a metalanguage. After you get used to the metalanguage, it will speed your understanding of the correct language syntax. The language syntax is described using the modified *Backus-Naur form* notational scheme. All the language elements and the rules for the combination of those elements are described.

A metalanguage cannot specify everything about a language; only its syntax. Readers already familiar with the syntax of BASIC should be careful not to assume that they know how a statement functions from just reading its syntax. Examples of this might include assumptions about the type of editing available when entering an input line, or the operation of the disk I/O statements. Chapter 8, IIGS "BASIC Reference," contains a detailed explanation of the all the variables, operands, operators, expressions, functions, statements, and commands in IIIIGS BASIC.

An element, expression, or statement is defined like this:

```
(element to be defined)
        ::= (some combination of defined elements)
```

Any uppercase letters or punctuation marks appearing on the second line after the definition assignment operator, '::=', must be entered exactly as shown. Lowercase letters or words represent variable information that you must fill in with a specific case. For example, in the following definition

```
WRITE# statement
        ::=     WRITE# filenum [, recnum] ; [expr[{, expr}] ]
```

the word WRITE and the '#' must be typed as shown, as must the commas and semicolon.

The filenum, recnum, and expr elements must be replaced with specific cases of the that type of element. Some definitions may have two or more lines each beginning with the definition assignment operator, ::=. These lines define variations for an element, expression, or statement.

The braces, { }, vertical bar, |, and brackets, [ ], are NOT to be entered, but instead are used to indicate how to combine the elements to construct variations of a statement or expression. These characters are elements of the metalanguage.

|	Separate alternative elements, one of which must be selected.
[ ]	Enclose optional elements.
{ }	Enclose a repeatable element or sequence of elements that Must occur at least once.
\ \	Enclose elements whose value are to be used

Here's an intuitive example of how this metalanguage is used to describe the various parts of Basic's syntax. In this example, we'll use houses, as they are familiar objects to most of us:

house

::= roof{door}{window}[fireplace][all-electric kitchen]

A house has a roof, one or more doors, one or more windows, and may have a fireplace and/or an all-electric kitchen.

home

::= house | cottage | mansion

A home can be a house, cottage, or mansion.

price

::= \house\

The selling price is the value of the house.

This notational scheme is known as modified BNF (Backus-Naur Form) after John Backus and Peter Naur, who first adapted it for use with computer languages. Its essential features were invented in ancient times by the Hindu scholar Panini, who was doing research into the grammar (syntax) of Sanskrit.

The first part of this appendix is a list of all elements used in BASIC statements. The second part describes the syntax of expressions. The third parts lists the various functions alphabetically by type. The last part of the appendix is an alphabetical listing of all BASIC statements and commands.

Elements

Discrete elements

uppercase
::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

lowercase
::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

foreign language char
::= [|\|]|^|{|||}|~ (special letters in foreign languages)

letter
::= uppercase|lowercase

digit
::= 0|1|2|3|4|5|6|7|8|9

line number|linenum
::= {digit} where value is in the range 1...65279 (1..$FF00-1)

line label|label
::= letter[{letter|digit|.}] (max length = 30 chars)

special
::= !|#|$|%|&|'|(|)|*|+|,|-|.|/|:|;|<|=|>|?|@|^|_
(NOTE: # becomes £, and/or @ becomes § in foreign languages
except french where @ becomes à)

character
::= uppercase|lowercase |digit|special|foreign language char

control character
::= ascii codes with hex values $01 ... $1F (NOTE: $00 is excluded)

prompt character
::=)

return
::= ascii carriage return (a control character=13 or $0D)

Operands

name
::= letter[{ letter|digit|. }] (maximum length = 30 chars)

subscript
::= (subsexpr [{,subsexpr}])

Note: the maximum range of any subscript is 0..32767 but the range for a given type of array is limited by the 1024K byte maximum array size and the number of dimensions.

variable name
::= name[@|#|$|%|&] (# => £, @ => § in most foreign languages)

array name
::= name[!|@|#|$|%|&]
(# => £, @ => § in most foreign languages)

unsigned integer
::= {digit} (size of integer determined by value as follows:)
(-32768 thru 32767 becomes a single integer)
(-2147483648 thru 2147483647 => a double integer)
(+/- 9223372036854775807 becomes a long integer)

not a number |nan
::= NaN[({digit})]

infinity
::= INF

exponent
::= E|e[+|-]{digit}

mixed constant
::= unsigned integer .
::= [unsigned integer].unsigned integer

finite constant
::= unsigned integer exponent
::= mixed constant [exponent]

unsigned real
::= finite constant|infinity|nan

single-precision constant (with 7 or fewer significant digits)
::= [+|-] unsigned real constant

double -precision constant (with 8 thru 15 significant digits)
::= [+|-] unsigned real constant

extended-precision constant (with 16 thru 20 significant digits)
::= [+|-] unsigned real constant

single-precision name|s-p name
::= name

single-precision variable|s-p variable
::= name[subscript]

double-precision name| d-p name
::= name# (name£ in some foreign languages)

double-precision variable|d-p variable
::= name# [subscript] (name£ [subscript] in some foreign languages)

real variable
::= s-p variable|d-p variable

single integer name|siname
::= name%

single integer variable|sivar
::= name% [subscript]

double integer name
::= name@ (name§ in most foreign languages)

double integer variable|divar
::= name@ [subscript] (name§ [subscript] in most foreign languages)

long-integer name
::= name&

long-integer variable|livar
::= name& [subscript]

integer variable|ivar
::= sivar|divar|livar

```
structure name
        ::=     name!

structure variable|stvar
        ::=     name! subscript

literal
        ::=     {character|control  character}          (max length = 255 chars)

string constant
        ::=     "literal"

null string
        ::=     ""

constant
        ::=     string constant|unsigned real |unsigned integer

string variable name
        ::=     name$

string variable|svar
        ::=     name$[subscript]

arithmetic variable|avar
        ::=     integer variable|real variable

variable|var
        ::=     arithmetic variable|string variable|integer variable

variable list|varlist
        ::=     var  [{,var}]

array variable
        ::=     name[ !|@|#|$|%|& ] subscript
                                        (# => £, @ => § in most foreign languages)
```

Reserved variables

```
reserved numeric variable|nresvar
        ::= AUXID@|ERR|ERRTOOL|EOF|ERRLIN|FRE|HPOS|INDENT|JOYY|
         KBD|LISTTAB|OUTREC|PDL9|SECONDS@|SHOWDIGITS|VPOS
```

```
reserved string variable | sresvar
        PREFIX$ | PROGNAM$ | DATE$ | TIME$

modifiable resvar | mresvar
    ::=  HPOS | VPOS | INDENT | LISTTAB | OUTREC | PREFIX$ | PROGNAM$
         SHOWDIGITS
```

Miscellaneous elements

```
filenum
    ::=  ubexpr (1.. 29 a maximum of 6 disk files open at one time)
                    ( with ProDOS-16 V2.0 a max of 31 files open)

dirname | filename
    ::=  \ UCASE$(letter[letter | digit | .] ) \ (max length = 15 for ProDOS)

volname
    ::=  /dirname

prefix selector | pfx
    ::=  0|1|2|3|4|5|6| |7|8

diskname
    ::=  .D digit | .D digit digit

chardevicename
    ::=  .CONSOLE | .PRINTER | .MODEM | .NULL | .NETPTR1 | .MEMBUFR |
         .filename

pathname
    ::=  [{dirname/}] filename                    (PREFIX$ applies)
    ::=  [volname[{/dirname}]/]filename
    ::=  [pfx[{/dirname}]/]filename

directory-pathname
    ::=  dirname{/dirname}[/]                     (PREFIX$ applies)
    ::=  diskname
    ::=  volname [{/dirname}][/]
    ::=  pfx[{/dirname}]/

open pathname
    ::=  chardevicename | pathname
```

filepath

::=	pathname		immediate mode
::=	sexpr	(with \pathname\)	deferred mode

dirpath

::=	directory-pathname		immediate mode
::=	sexpr	(with \directory-pathname\)	deferred mode

openpath

::=	open pathname		immediate mode
::=	sexpr	(with \open pathname\)	deferred mode

ftyp

::= a three character filetype abbreviation as defined in Appendix J.

wildcardchar

::= =|-|#

wildcardfilnam

::= \ UCASE$(letter|wildcardchar[wildcardchar|letter|digit|.]) \
(maximum length of 15 characters for ProDOS)

function name

::= name[@|%|#|&]

Procname

::= name

library name

::= name (maximum length = 20 characters)

recnum

::= aexpr (result range limited to 0..8,388,607)

recsize

::= aexpr (result range limited to 1..32767)
(except BDF files are limited to 3..32767)

repeat factor|rpt

::= {digit} (result limited to 1... 255)

string spec

::= {[rpt]A|C|R}

A reserves a character position in a left-justified string.
C reserves a character position in a centered string.
R reserves a character position in a right-justified string.

literal spec

::= {[rpt]X | / | string constant}

An X character prints a space.
A / character prints a carriage return.
A string constant prints literal text. When a repeat factor is used with a string constant the literal is repeated.

digit spec

::= [rpt] [{# | Z | &}] . [rpt]{# | Z | &}
::= [rpt]{# | Z | &}[.]

#	reserves one numeric digit position; leading zeros are replaced with spaces.
Z	reserves one numeric digit position; leading zeros are printed.
&	reserves one position for a numeric digit or a comma; commas are inserted after every third digit starting at the decimal point and working left. Commas are included in the character count and leading zeros are replaced with spaces. A minimum of 5 &'s must occur to left of the decimal point.

fixspec

::= [**][$][+ | -] digitspec
::= [**][+ | -][$] digitspec
::= [**][$] digitspec [+ | -]
::= $$[+ | -] digitspec
::= $$ digitspec [+ | _]
::= [++ | --][$] digitspec

+	reserves a character position for printing the sign.
-	reserves a character position for printing the sign; a - is printed for a negative value, a space for positive value.
$	reserve a character position for a dollar sign.
* *	prints asterisks instead of spaces in unused leading positions.
++	reserves the rightmost unused position for the sign (and a following dollar sign if any).
--	same as ++ above, except the sign is replaced by a space when the value is positive.
$ $	reserves the rightmost unused position for a dollar sign (and a following numeric sign if any). NOTE: $$, ++, and -- may not be used if the digit spec contains a Z.

scipart

::= [rpt]# | Z

fracpart
::= . [rpt][{# | Z}]

expspec
::= EEE | EEEE | EEEEE | EEEEEE | 3E | 4E | 5E | 6E

scispec
::= [+ | -][scipart][fracpart] expspec

engrpart
::= ### | ZZZ

engrspec
::= [+ | -] engrpart[fracpart] expspec

spec
::= string spec | literal spec | fixspec | scispec | engrspec

Functions

::=

CHR$ function
::= CHR$(ubexpr)

CONV$ function
::= CONV$(expr)

ERRTXT$ function
::= ERRTXT$(ubexpr)

EXFN function
::= EXFN$ name [(lexpr{,lexpr})]

HEX$ function
::= HEX$(aexpr) (value range limited to +/-2^32-1)

INSTR function
::= INSTR(sexpr,sexpr[,ubexpr])

LEFT$ function
::= LEFT$(sexpr,ubexpr)

MID$ function
        ::=   MID$(sexpr,ubexpr1 [,ubexpr2])

PFX$ function
        ::=   PFX$(pfx)

RIGHT$ function
        ::=   RIGHT$(sexpr,ubexpr)

REP$ function
        ::=   REP$(sexpr,ubexpr)

SPACE$ function
        ::=   SPACE$(ubexpr)

STR$ function
        ::=   STR$(aexpr)

UCASE$ function
        ::=   UCASE$(sexpr)

Arithmetic functions

::=

ABS function
        ::=   ABS(aexpr)

ANNUITY function
        ::=   ANU(aexpr,aexpr)

ATN function
        ::=   ATN(aexpr)

COMPOUND function
        ::=   COMPI(aexpr,aexpr)

CONV function
        ::=   CONV[@|#|%|&](expr)

COS function
        ::=   COS(aexpr)

EXP function
        ::=   EXP(aexpr)

EXP1 function

```
        ::=     EXP1(aexpr)                 (base e Exponential minus 1)
```

EXP2 function

```
        ::=     EXP2(aexpr)                 (base 2 Exponential)
```

FIX function

```
        ::=     FIX(aexpr)
```

INT function

```
        ::=     INT(aexpr)
```

LOG function

```
        ::=     LOG(aexpr)
```

LOGB% function

```
        ::=     LOGB%(aexpr)                (binary exponent of aexpr)
```

LOG1 function

```
        ::=     LOG1(aexpr)                 (base e log of aexpr+1)
```

LOG2 function

```
        ::=     LOG2(aexpr)                 (base 2 logarithm)
```

NEGATE function

```
        ::=     NEGATE(aexpr)
```

RND function

```
        ::=     RND(aexpr)
```

ROUND function

```
        ::=     ROUND(aexpr)
```

SCALB function

```
        ::=     SCALB(siexpr,aexpr)         (aexpr*2^siexpr)
```

SGN function

```
        ::=     SGN(aexpr)
```

SIN function

```
        ::=     SIN(aexpr)
```

SQR function
::= SQR(aexpr)

TAN function
::= TAN(aexpr)

Miscellaneous functions

::=

ASC function
::= ASC(sexpr)

BASIC@ function
::= BASIC@(ubexpr)

BTN function
::= BTN(ubexpr)

DATE function
::= DATE(ubexpr)

EOFMARK function
::= EOFMARK(filenum)

EXEVENT@ function
::= EXEVENT@(ubexpr)

EXFN function
::= EXFN[%|@|#|&|$][_]name [(lexpr{,lexpr})]

FILE function
::= FILE(sexpr[, FILTYP= DIR|TXT|SRC|BDF|ubexpr])

FILTYP function
::= FILTYP(filenum)

FREMEM function
::= FREMEM(ubexpr)

JOYX function
::= JOYX(ubexpr)

LEN function
::= LEN(sexpr)

PDL function

::= PDL(ubexpr)

PEEK function
::= PEEK(iexpr) (range limited to 0..2^24)

R.STACK% function
::= R.STACK%(ubexpr)

R.STACK@ function
::= R.STACK@(ubexpr)

R.STACK& function
::= R.STACK&(ubexpr)

REC function
::= REC(filenum)

TASKREC% function
::= TASKREC%(ubexpr)

TASKREC@ function
::= TASKREC@(ubexpr)

TEN function
::= TEN(sexpr)

TIME function
::= TIME(ubexpr)

TYP function
::= TYP(filenum)

UBOUND function
::= UBOUND (array-name[()] [,ubexpr])

UIR function
::= UIR(ubexpr)

VAR function
::= VAR(stvar,vtype[,lgth])

VAR$ function
::= VAR$(divar[,lgth])

VARPTR function
::= VARPTR(variable)

VARPTR$ function
::= VARPTR$(string variable)

Print functions

::=

SPC function
        ::=   SPC(aexpr)

SCALE function
        ::=   SCALE(siexpr,expr)

TAB function
        ::=   TAB(aexpr)

Numeric functions

numeric function
        ::=   arithmetic function | miscellaneous function

Expressions

relational operator | relop
        ::=   = | < | > | <> | >= | => | <= | =< | <=>

string term | sterm
        ::=   svar | string function | string constant | (string expression)

string expression | sexpr
        ::=   sterm {+ sterm}

arithmetic factor
        ::=   unsigned real | unsigned integer
        ::=   arithmetic variable | resvar
        ::=   numeric function call
        ::=   (arithmetic expression)
        ::=   [NOT][+ | -] arithmetic factor

arithmetic term
        ::=   arithmetic factor {^ | * | / | DIV | MOD | + | - arithmetic factor}
                    (operators shown in order of precedence, highest first)

relational term
::= arithmetic term [{relop arithmetic term}]
::= sexpr {relop sexpr}

logical term
::= relational term {AND | OR | XOR relational term}
(operators shown in order of precedence, highest first)

arithmetic expression
::= arithmetic term
::= relational term (integer result = 0 for false, 1 for true)
::= logical term (integer result = 0 for false, 1 for true)

unsigned byte expression | ubexpr
::= aexpr (with resulting range of 0 ... 255)

subscript expression | subsexpr
::= aexpr (with result range of 0 ... 32767)

linenum expression | lnexpr
::= linenum | label

single integer expression | siexpr
::= aexpr (with result range of -32768...0...32767)

integer expression | iexpr
::= aexpr (integer with range of -2^63...0...2^63)

expression | expr
::= aexpr | sexpr

Statements

Statement form

Statement List
::= statement [{: statement}]

Immediate Statement | command
::= statement list return (max of 239 characters)

Deferred Statement | program line
::= linenum [label:]statement list return (max of 239 characters)

Statement definitions

ASSIGN statement
::= ASSIGN chardevicename | sexpr,slot [,AUTO]

BREAK statement
::= BREAK ON | OFF

CALL statement
::= CALL libname [(lexpr{,lexpr})]
::= _libname [(lexpr{,lexpr})]

CALL% statement
::= CALL% ubexpr1,ubexpr2,ubexpr3 [(lexpr{,lexpr})]

CATALOG statement
::= CATALOG [dirpath] (displayed in 80 columns)
::= CAT [dirpath] (displayed in 40 columns)

CHAIN statement
::= CHAIN filepath[,lnexpr]

CLEAR statement
::= CLEAR
::= CLEAR aexpr
::= CLEAR LIBRARY
::= CLEAR INVOKE

CLOSE statement
::= CLOSE[# filenum]

COPY statement
::= COPY filepath1 ,filepath2 [,linenum]

CREATE statement
::= CREATE filepath ,FILTYP= DIR | TXT | SRC | BDF | ubexpr[,recsize]

DATA statement
::= DATA [literal | constant][{literal | constant}]

DATE$ statement
::= DATE$ =ubexpr,ubexpr,ubexpr (YY,MM,DD)

DEF FN statement
::= DEF FN function name(varlist) = aexpr
or
::= DEF FN function name(varlist): [statement list]
...
END FN function name
or
::= DEF PROC procname[(varlist)]: [statement list]
...
END PROC [procname]

DELETE statement
::= DELETE filepath

DIM statement
::= DIM array variable name[{,array variable name}]

DIR statement
::= DIR [dirpathname [/wildcardfilnam [,[-]ftyp[{,ftyp}]]

DO statement
::= DO (used as DO ... UNTIL or DO ...WHILE...UNTIL)

ELSE statement
::= ELSE [lnexpr|statementlist] (only as 1st statement in a line)

END statement
::= END
::= END PROC [procname]
::= END FN function name

ERASE statement
::= ERASE varname|array variable name()
[{,varname| array variable name()}]

ERROR statement
::= ERROR ubexpr

EVENTDEF statement
::= EVENTDEF ubexpr{,linnum|label}

EXCEPTION statement
::= EXCEPTION {ON|OFF|0}

EXEC statement
::= EXEC filepath,[OFF] (filetype must be TXT|SRC)
::= EXEC # filenum,[OFF] (chains to previous EXEC)

FN let statement

```
        ::=     FN function name=expression
        ::=     FN localvar=expression
```

FOR statement

```
        ::=     FOR arithmetic variable = aexpr1 TO aexpr2 [ STEP aexpr3]
```

GET# statement

```
        ::=     GET #filenum[,length[,recnum]] ; stvar
```

GET$ statement

```
        ::=     GET$ [#filenum[,recnum]; svar
```

GOSUB statement

```
        ::=     GOSUB lnexpr
```

GOTO statement

```
        ::=     GOTO lnexpr
```

GRAF statement

```
        ::=     GRAF INIT mode                          (mode = 0|320|640)
        ::=     GRAF ON
        ::=     GRAF OFF
```

HOME statement

```
        ::=     HOME
```

IF statement

```
        ::=     IF lexpr THEN lnexpr|statementlist [ else statement ]
        ::=     IF lexpr GOTO lnexpr[ else statement ]
        or
        ::=     linnum IF lexpr THEN lnexpr|statementlist
                linnum else statement
        or
        ::=     linnum IF lexpr
                linnum then statement
                linnum else statement
```

IMAGE statement

```
        ::=     IMAGE spec[{,spec}]
```

INIT statement

```
        ::=     INIT diskname,volname                           immediate mode
                    (prompts user in immediate mode if volume already exists)
        ::=     INIT sexpr,sepxr                                deferred mode
                    (NO USER PROMPT in deferred mode)
```

INPUT statement
::= INPUT [string constant,|;] varlist

INPUT# statement
::= INPUT# filenum[,recnum][; varlist

INPUT USING statement
::= INPUT USING lnexpr; string variable

INVERSE statement
::= INVERSE

INVOKE statement
::= INVOKE
::= INVOKE filepath[{,filepath}]
::= INVOKE APPEND filepath[{,filepath}]

LET statement
::= [LET] var|mresvar = expression
(multi-line function references allowed in LET ONLY)

LIBFIND statement
::= LIBFIND sexpr,sivar1,sivar2,sivar3

LIBRARY statement
::= LIBRARY
::= LIBRARY filepath[{,filepath}]
::= LIBRARY APPEND filepath[{,filepath}]

LOAD statement
::= LOAD filepath

LOCAL statement
::= LOCAL varname{,varname} (only inside DEF...END)

LOCK statement
::= LOCK filepath

MENUDEF statement
::= MENUDEF ubexpr{,linnum|label}

NEW statement
::= NEW

NEXT statement
::= NEXT [control variable [,{control variable}]

NORMAL statement
::= NORMAL

NOTRACE statement
::= NOTRACE

OFF EOF# statement
::= OFF EOF# filenum

OFF ERR statement
::= OFF ERR

OFF EXCEPTION statement
::= OFF EXCEPTION

OFF TIMER statement
::= OFF TIMER

OFF KBD statement
::= OFF KBD

ON BREAK statement
::= ON BREAK statementlist

ON EOF# statement
::= ON EOF# filenum statement list

ON ERR statement
::= ON ERR statement list

ON EXCEPTION statement
::= ON EXCEPTION statement list

ON GOSUB statement
::= ON aexpr GOSUB lnexpr[{,lnexpr}]

ON GOTO statement
::= ON aexpr GOTO lnexpr[{,lnexpr}]

ON KBD statement
::= ON KBD statement list

ON TIMER statement
::= ON TIMER (aexpr) GOSUB linenum | label

OPEN statement
::= OPEN openpath ,[FILTYP=DIR | TXT | SRC | BDF | ubexpr][FOR INPUT |

```
                    OUTPUT|APPEND|UPDATE],AS # filenum [,recsize]
        ::=     OPEN # filenum ,FOR INPUT|OUTPUT|UPDATE AS # filenum
[,recsize]

OUTPUT statement
        ::=     OUTPUT# filenum

PERFORM statement
        ::=     PERFORM libname [ (lexpr[{,lexpr }] ) ]

POKE statement
        ::=     POKE iexpr,ubexpr               (iexpr limited to 0..2^24)

POP statement
        ::=     POP

PREFIX statement
        ::=     PREFIX [pfx,][dirpath]                  (except pfx=8)

PRINT statement
        ::=     ?|PRINT {[,|;]lexpr]}[,|;]

PRINT# statement
        ::=     ?|PRINT # filenum[,recnum] [; expr [{; expr}]] [;]

PRINT USING statement
        ::=     ?|PRINT USING lnexpr|string|svar ; [ expr [{, expr}]] [;]

PRINT# USING statement
        ::=     ?|PRINT # filenum [,recnum] USING
                 linenum|string|svar ; [ expr [{, expr}]] [;]

PUT statement
        ::=     PUT # filenum[,[recsize][,recnum]];stvar

QUIT statement
        ::=     QUIT [filepath]

RANDOMIZE statement
        ::=     RANDOMIZE aexpr

READ statement
        ::=     READ varlist

READ# statement
        ::=     READ# filenum [,recnum] [; varlist]
```

```
REM statement
        ::=     REM literal

RENAME statement
        ::=     RENAME filepath1,filepath2[,FILTYP= DIR|TXT|SRC|BDF|ubexpr]

RESTORE statement
        ::=     RESTORE [lnexpr]

RESUME statement
        ::=     RESUME [NEXT|COPY]

RETURN statement
        ::=     RETURN
        ::=     RETURN 0
        ::=     RETURN linnum

RUN statement
        ::=     RUN [filepath [,lnexpr]
        ::=     RUN [lnexpr]

SAVE statement
        ::=     SAVE [filepath]
        ::=     SAVE AS
        ::=     SAVE AS filepath

SET statement
        ::=     SET (stvar,[siexpr]) = expr
        ::=     SET (stvar,[siexpr]) = ^sexpr
        ::=     SET (stvar,[siexpr]) = * iexpr[,length]

STOP statement
        ::=     STOP

SUB$ statement
        ::=     SUB$ (svar,ubexpr [,ubexpr]) = sexpr

SWAP statement
        ::=     SWAP var1,var2

TASKPOLL statement
        ::=     TASKPOLL INIT iexpr,iexpr
        ::=     TASKPOLL ON
        ::=     TASKPOLL OFF

TEXT statement
        ::=     TEXT
```

TEXTPORT statement
::= TEXTPORT ubexpr1,ubexpr2 TOubexpr3,ubexpr4

THEN statement
::= THEN statement list (only as 1st statement in a line)

TIME$ statement
::= TIME$ =ubexpr,ubexpr,ubexpr (HH,MM,SS)

TIMER statement
::= TIMER ON|OFF

TRACE statement
::= TRACE [TO # filenum]

TYPE statement
::= TYPE filepath [TO #filenum]

UNLOCK statement
::= UNLOCK pathname|sexpr

UNTIL statement
::= UNTIL [lexpr] (must be paired with WHILE or DO)

WHILE statement
::= WHILE [lexpr] (must be paired with UNTIL, may follow DO)

WRITE# statement
::= WRITE# filenum[,recnum] [; expr [{, expr}]]

Commands

The following commands may only be executed in immediate mode and are **not** supported by the Run-time version of the interpreter nor will they likely be supported by a IIGS BASIC compatible compiler, should one become available in the future.

AUTO command
::= AUTO [linenum [,increment]]

CONT command
::= CONT

DEL command
::= DEL linenum1 [,|- linenum2]

EDIT command

::= EDIT linenum1 [,|- linenum2]

HLISTcommand

::= HLIST [linenum1] [,|- [linenum2]]

LISTcommand

::= LIST [linenum1] [,|- [linenum2]]

RENUM command

::= RENUM [newlinenum][,[increment] [,linnum1 [-linnum2]]]

Appendix H

Appendix Needs Title

The TDFBUILD utility

The TDFBUILD utility creates IIGS BASIC toolbox definition files (TDFs) from ordinary text (TXT) files. Source records for TDFBUILD are simple text lines of arguments terminated by carriage returns. There are two types of records: header records for each tool set in the TDF and definition records. Each definition record provides the complete interface definition for a single tool set function call.

Each definition record includes the function name (up to 20 characters), tool set number, function number, result stack size (in words), error-handling mode, function result type, input argument count, and the type for each input argument.

The output TDF is a hashed symbol table with variable length binary records, one per interface definition. The symbol table begins with a length word followed by a 32-word hash thread pointer table, one or more 25 byte tool set information records, and then one or more variable-length binary interface-definition records. The TDF format is defined later in this appendix.

TDF source record format

Input header records

The format of the input header record is as follows:

Rectype,Loadflag,Version,Tool#,0,Toolname

The parameters and options are described below.

Table H-1

Parameter	Option	Description
Rectype	H	Header record
Loadflag	L	Load the tool with Tool Locator LOAD1TOOL call
	N	Don't load the tool with LOAD1TOOL (ROM tools)
Version	$*nnnn*	Version of the tool set used with LOAD1TOOL call
Tool#	999	Toolbox number of the tool set (1 through 255)
spare	0	spare parameter; must be zero
Toolname	1 to 20 chars	Tool set name, such a Window Manager

Options shown as capital letters must be entered as shown. *Version* is entered as a hex number and must begin with a dolar sign ($). *Toolname* is not used by IIgs BASIC during execution, but it is required. Any characters after the first 20 in *Toolname* (counting blanks) are ignored by TDFBUILD. We suggest that you maintain the input file revision date and level in this portion of the header record, using it as an assembler comment field.

Interface definition records

The interface definition record is more complex. It has the following format:

Type,Tool#,Func#,RSize,ErrMd,RType,P_{Cnt}*[*PT_1*,*PT_2*, ... *PT_{Pcnt}*]Functionname*

This format is described below.

Table H-2

Parameter	Char	Description
Type	P	Procedure called via Tool Locator interface (result type = N)
	F	Function called via Tool Locator interface (result type <> N)
Tool#	digits	A tool set number 01 to 255; leading zeros not required
Func#	digits	A function number 01 to 255; leading zeros not required
RSize	digits	A word count of the number of words returned on the stack.
ErrMd	N	No error result returned: R.STACK%(0) = 0000
	C	Carry returned as integer function result: R.STACK%(0) = 000c
	X	Carry set := error; return X register as result: R.STACK%(0) = X-reg
	A	Carry set := error; return A register as result: R.STACK%(0) = A-reg

	B	Carry set := error; return C register as result: R.STACK%(0) = C-reg
	E	Carry set := error in X-register dispatch via BASIC ON ERR
	P	Carry set := error in A-register dispatch via ProDOS error translator then to ON ERR with ProDOS CALL error
	T	Carry set := error in C-register dispatch via TOOLSET error save then to ON ERR with TOOLSET CALL error
RType	N	Nonspecific result type with ID_RSIZ 0 through 32 bytes (if RSIZ = 0 then R.STACK%(0) = Carry, (1)=A,(2) = X,(3) = Y
	I	Single-integer result (2 bytes)
	A	Address or pointer (24 lsbits of 32) (4 bytes)
	L	Double-integer result (4 bytes)
	C	Long-integer result (8 bytes)
	F	Single-precision real result (4 bytes)
	D	Double-precision real result (8 bytes)
	X	Extended-precision expression result (10 bytes)

The following three function result types cause conversion into a BASIC string and construction of a BASIC temporary string descriptor.

T	3-byte pointer + length byte (msbyte) (4 bytes)
P	Pointer to a counted string result (4 bytes)
Z	Pointer to C string (0 ended) result (4 bytes)

❖ *Note:* String functions should always return a string result (null or otherwise). The only usable error mode for a string function is option E, dispatch X to BASIC. You cannot use any other error mode because returning an integer (error code) for a string result always generates a type mismatch error.

P_{cnt}	digits	A count of the number of parameters in the range of 0 to 10.
PT	I	Single-integer parameter (2 bytes)
	A	Address or pointer parameter (4 bytes)
	L	Double-integer parameter (4 bytes)
	C	Long-integer parameter (8 bytes)
	F	Single-precision real parameter (4 bytes)
	D	Double-precision real parameterb(8 bytes)
	X	Extended-precision expression result (10 bytes)
	P	Pointer to a counted string parameter (4 bytes) (causes automatic conversion from a BASIC string to a p-string with a maximum length of 254 bytes)
	S	Pointer to a BASIC string descriptor (4 bytes)

Functionname 1 to 20 characters function name, including letters (A-Z) and digits

The interface definition record format will be converted into a prehashed binary symbol table format that will maximize search speed of the *Functionname* parameter. A simple yet effective equivalence class hashing algorithm (32 classes) keying on the characters in the function name will be used. The TDF will include a head pointer table for the hash threads and an information record for each tool set defined in the file.

The library dictionary in memory will consist of a linked list of tables, one table per prehashed TDF. The search process will require search if only 1 of 32 link-list threads in each TDF table until a match is found. The order of the linked list will be controlled by the order in which the LIBRARY statement loads the TDF. A separate pseudo-TDF table will be used for invoked module entry points.

TDF Format

The output of the TDFBUILD utility is a TDF. The binary format of a TDF has been designed to match exactly the format of a TDF table in memory, as used by IIGS BASIC in the library dictionary. The LIBRARY statement loads a TDF into the library segment, without any conversion or modification of its content.

The library segment also includes a partition that contains one TDF table for all the invokable modules. This TDF table is extended dynamically each time an invokable module is loaded. Each invokable module contains a private data segment named DICTIONARY, which has a header record and up to 255 entry point definition entries in the same format shown here. (See Appendix I, "Invokable Modules," for details.)

The format of a TDF is described below.

Table H-3

Field	Size	Description
Length	1 word	Total length of all data in the file
HeadTbl	32 words	Hash-thread link (offset) table
InfoTbl	25 bytes	Tool set information table
ID.Entry	15-45 bytes	Interface definition entry

The HeadTbl contains 32 words; each word is the beginning of a equivalence class hashed linked list. The hashing algorithm is implemented by the invokable module HASHTDF.INV. The algorithm is a slightly modified 8-bit LRC of the characters of the function's name. The hash key that indexes the HeadTbl is the least significant 5 bits of the modified 8-bit LRC.

The hash-thread link table

Each word in the HeadTbl contains a self-relative offset to the next entry in that thread, if any. In other words, if a pointer is pointing at the low byte of the HeadTbl word, the address of the next entry in the thread is created by adding the offset in the word to the pointer. Likewise, each actual interface definition entry begins with a word that contains the offset from itself to the next entry, or a zero for the last entry in the thread.

The Tool set information table

The InfoTbl is a fixed length table with four fields. It describes a tool set and provides the parameters required by the Tool Locator LOAD1TOOL call. A TDF may have all the interface definitions for one or more tool sets or a mixture of partial interface definitions for a number of tool sets, as suits the needs of the programmer.

The format of the tool set information table is described below.

Table H-4

InfoTbl Item	Size	Description	
Load flag	1 byte	$00	Don't issue LOAD1TOOL call
		$80	Last InfoTbl record; don't issue LOAD1TOOL call
		$01	Issue LOAD1TOOL call using version number and tool numbers
		$81	Last InfoTbl record; issue LOAD1TOOL call
Tool version	1 word		Tool set version number used in calling LOAD1TOOL
Tool set number	1 word		Tool set tool number used in calling LOAD1TOOL (valid values are $0001 through $00FF)
Toolset name	20 characters		Title of the tool set (for example, Window Manager)

The tool set interface definition entry

The interface definition entry is designed to support toolbox calls, as well as invokable module entry-point interface definitions. Some bytes within the entry serve different purposes for these two functionss so the table is defined twice, first showing the fields used for tool set interface definitions and again for invokable module inteface definitions.

The format of the tool set interface definition entry is decribed below.

Table H-5

Field name	Size	Description
ID_NXT	Word	Self-relative offset to next thread entry
ID_ELEN	Byte	Length of this entry (all bytes)
ID_ENTRY	Word	Unused by tool set interface definitions; must be zero
ID_TOOL	Byte	Tool set number for this function or procedure
ID_FUNC	Byte	Tool function number for this function or procedure
ID_RSIZ	Byte	Result stack size in words (0 through 16)
ID_ERRM	Byte	Error handling mode
	$00	N = No Error result is returned
	$02	E = carry set := X-register(8-bit) is BASIC error #
	$04	P = carry set := A-register(8-bit) is ProDOS error #
	$06	T= carry set := C-register(16-bit) is tool set error #
	$08	C = carry set := Return carry as R.STACK%(0) = 0/1
	$0A	X = carry set := Return X-register(8-bit) as R.STACK%(0)
	$0C	A = carry set := Return A-register(8-bit) as R.STACK%(0)
	$0E	B = carry set := Return C-register(16-bit) as R.STACK%(0)
ID_TYP	Byte	Type of entry point in tool set
	$81	Procedure (ID_FNTYP = N; ID_RSIZ usually zero)
	$82	Function (ID_FNTYP <> N; ID_RSIZ nonzero)
ID_FNTYP	Byte	Type of function result (input spec calls this Rtype)
	$00	N = Indicates untyped (multiple) result or no result; for multiple results, ID_RSIZ will be nonzero; user must use R.STACK functions to obtain the results. For ID_RSIZ= to zero the return registeristers are available: R.STACK%(0) = Carry, (1) = A-register, (2) = X-register,(3) = Y-register
	$01	I = Integer result (a word)
	$02	L = Double integer (a long)
	$02	A = Address or pointer (24 least significant of 32 bits)
	$03	C = Comp integer or BASIC long integer
	$04	F = SANE single-precision real result
	$05	D = SANE double-precision real result
	$06	X = SANE extended-precision real result
	$07	T = String descriptor (convert 3-byte pointer + length byte)
	$0F	P = Pointer to a counted string result (convert it)
	$17	Z = Pointer to a C-string (0-ended) result (convert it)
ID_PCnt	Byte	Input parameter count (0 to 10)
ID_POffset	Byte	Offset from first byte of ID.Entry to PT1 item below
ID_NLen	Byte	Length of ID_NAME item (1 to 20 characters)
ID_NAME	1..20	Name of the procedure or function (A-Z, 0-9 only)

ID_PT1	Byte	Parameter type for input argument 1
ID_PT2..10	Bytes	Parameter type byte(s) for input Arguments 2 through 10
	$01	I = Integer argument (a word)
	$02	L = Double-integer argument (a long)
	$02	A = Address argument (24 least significant of 32 bits)
	$03	C = Comp integer argument (BASIC long integer)
	$04	F = SANE single-precision real argument
	$05	D = SANE double-precision real argument
	$06	X = SANE extended-precision real argument
	$07	S = Address of BASIC string descriptor argument
	$87	P = Pointer to a counted string argument created from a BASIC string (descriptor address)

The TDFBUILD utility will request an input pathname, read 1 to N header records from the input file (creating an InfoTbl for each one), and then process interface definition records until end of the file. You can build the data in a structure array, and then create an output file (with the total length as record size) and write a single record with PUT# (and the length option), after calling the HASHTDF invokable module to hash the entries.

Using the HASHTDF.INV invokable module

An invokable module, HASHTDF.INV, is provided to perform the hashing of an otherwise complete TDF formatted data structure. The input content for a HASHTDF.INVis in a slightly different format than it is in a TDF.

HASHTDF.INV expects ID_NXT of each entry to contain the total length of the entry (including ID_NXT itself). HASHTDF.INV also expects the HeadTbl to be all zeros. The first word of the TDF table must contain the total length of the data structure. HASHTDF.INV is called by PERFORM, with the entry point name HASHTDF and a single parameter. The parameter is the address of the beginning of the TDF data structure, that is, the address of the total length word.

HASHTDF.INV skips over the InfoTbl records and processes the interface definition entries in order. The length byte in ID_NXT is copied into ID_ELEN, and the ID_NXT field is filled in as the entries are threaded together in their respective equivalence classes. Upon return from HASHTDF, the data structure is ready to be written to disk as a completed TDF format file.

Invokable module interface definition entries

The interface definition entries for all the invokable modules are combined into a single TDF table by INVOKE. Each invokable module contains a private data segment named DICTIONARY, (the format is defined in Appendix I), and these segments are appended to the invoke definition table as each module is processed.

The format of an invokable module is described below.

Table H-6

Field name	Size	Description
ID_NXT	Word	Self-relative offset to next thread entry
ID_ELEN	Byte	Length of this entry (all bytes)
ID_ENTRY	Word	Entry-point offset (in memory) within the static code segment #001 of an invoked load file.
ID_TOOL	Byte	INVTAB offset to Loadseg information record (low byte)
ID_FUNC	Byte	INVTAB offset to Loadseg information record (high byte)
ID_RSIZ	Byte	Result stack size in words (0 through 16)
ID_ERRM	Byte	Error-handling mode
	$00	N = No Error result is returned
	$02	E = Carry set := X-register(8-bit) is BASIC error #
	$04	P = Carry set := A-register(8-bit) is ProDOS error #
	$06	T = Carry set := C-register(16-bit) is tool set error #
	$08	C = Carry set := Return carry as R.STACK%(0) = 0/1
	$0A	X = Carry set := Return X-register(8 bit) as R.STACK%(0)
	$0C	A = Carry set := Return A-register(8 bit) as R.STACK%(0)
	$0E	B=Carry set := Return C-register(16 bit) as R.STACK%(0)
ID_TYP	Byte	Type of entry point in the invoked module
	$01	Procedure (ID_FNTYP = N; ID_RSIZ usually zero)
	$02	Function (ID_FNTYP <> N; ID_RSIZ non-zero)
ID_FNTYP	Byte	Type of function result
	$00	N = Indicates untyped (multiple) result or no result; for multiple results, ID_RSIZE will be nonzero; user must use R.STACK to obtain the results. For ID_RSIZ = to zero the return registeristers are available: R.STACK%(0)=Carry, (1)=A-register, (2)=X-register,(3)=Y-register
	$01	I = Integer result (a word)
	$02	L = Double-integer (a long)
	$02	A = Address or pointer (24 least significant of 32 bits)
	$03	C = Comp integer or BASIC long integer
	$04	F = SANE single-precision real result
	$05	D = SANE double-precision real result
	$06	X = SANE extended-precision real result
	$07	T = String descriptor (convert 3-byte pointer + length byte)
	$0F	P = Pointer to a counted string result (convert it)
	$17	Z = Pointer to a C-string (0-ended) result (convert it)
ID_PCnt	Byte	Input parameter count (0 to 10)
ID_POffset	Byte	Offset from first byte of ID.Entry to PT1 item below
ID_NLen	Byte	Length of ID_NAME item (1 to 20 charaters)
ID_NAME	1 through 20	Name of the procedure or function (A-Z, 0-9 only)
ID_PT1	Byte	Parameter type for input argument 1

ID_PT2..10	Bytes	Parameter type byte(s) for input arguements 2 through 10.
	$01	I = Integer argument (a word)
	$02	L = Double-integer argument (a long)
	$02	A = Address argument (24 least significant of 32 bits)
	$03	C = Comp integer argument (BASIC long integer)
	$04	F = SANE single-precision real argument
	$05	D = SANE double-precision real argument
	$06	X = SANE extended-precision real argument
	$07	S = Address of BASIC string descriptor argument
	$87	P = Pointer to a counted string argument created from a BASIC string (descriptor address)

Appendix I

Invokable Modules

Introduction

The IIGS BASIC statements INVOKE and PERFORM give you an easy-to-use symbolic interface to assembly-language procedures and functions. This interface allows you to extend the capabilities of IIgs BASIC for specific appplications.

In this appendix, it is assumed that you have read and understood the sections on INVOKE, PERFORM, and EXFN in chapters 7 and 8 of this manual. In addition, you should be familiar with the Apple IIgs Programmer's Workshop (APW) Assembler.

The INVOKE statement creates interface definition table entries to be used by the PERFORM statement by reading a data segment, named DICTIONARY, from the invokable load file. Then INVOKE calls the ProDOS 16 System Loader to load (and relocate) the executable subroutines. Procedures are called from BASIC with PERFORM, and functions are called with EXFN.

The interface definition table provides the symbolic entry point names for the assembly-language procedures and functions. The interface definition also defines the input parameter count and type for each parameter, as well as the function result type, result stack size, error-handling mode, and the entry point offset in the executable code.

The APW Assembler provides the necessary tools for programmers to create a single code segment with up to 255 separate entry points. In addition to the actual code segment, an invokable module must include a private data segment named DICTIONARY.

The DICTIONARY segment is created according to the format described in this appendix, using DC directives. All the requirements for creating an invokable module can be met without resorting to linked commands of the Advanced Linker.

The PERFORM statement

The PERFORM statement hashes the name following the word PERFORM and searches the interface definition table (IDT) for the entry point name. The IDT entries are divided into 32 equivalence classes to reduce search time. Each equivalence class has a separate linked list to search, with the hash key for each class being derived from all the characters of the procedure name.

After a matching entry is found in the IDT, the parameter list requirements are used to process the argument list, if one is required. Each argument is type-checked against the matching parameter type, in the required order. If a string argument is given for a numeric parameter or vice versa, the message

```
?ARGUMENT TYPE MISMATCH ERROR
```

appears. If the argument is numeric but not the correct type, the argument is converted to the required type, using the appropriate CONV function, before pushing the argument on the CPU stack.

The argument list must have the proper number of parameters of the correct type in the proper order. If too many or too few arguments are present, the message:

```
?ARGUMENT COUNT ERROR
```

is displayed. After the arguments are all pushed on the stack, the run-time entry point address is calculated by adding the entry point offset, obtained from the IDT, to the load seg address for the code segment. The load seg address is saved in a separate table, called the invoke segment table (IST). This IST entry retains the System Loader information about the loaded code segment. The invokable subroutine is then entered as if a JSL instruction were executed.

After the procedure returns to BASIC, via an RTL instruction, any results are pulled off the CPU stack and saved in a buffer, called the return stack. A procedure may or may not return any results. Execution proceeds with the next statement in the BASIC program following the PERFORM statement. If there are any results in the return stack, you can access them by using the R.STACK% and R.STACK@ functions.

If the procedure returns with the processor carry flag set, an error condition is signaled, and it will be processed according to the error-handling mode specified in the IDT entry for the procedure. A number of useful options are available for different kinds of error handling, as described later in this appendix.

The EXFN function

EXFN (which stands for external function) is almost identical to PERFORM, except that a function type entry point must be found in the interface definition table. A function entry defines the number of function result words that must be pushed on the stack just before the arguments are pushed.

A function entry in the IDT also defines the type of the numeric or string result. Functions may return integers; double integers; long integers; and single-, double-, or extended-precision SANE real numbers as results. Three types of string results may be returned; all types are converted into a BASIC format string, with a temporary string descriptor, and the data are copied into the BASIC string literal pool.

The INVOKE statement

The INVOKE statement accomplishes the following functions:

1. Optionally deletes all previous invoked modules and recovers the memory used by those modules and their interface definitions.
2. Builds or extends the IDT (from the load file DICTIONARY segment) to allow symbolic access through the PERFORM statement or the EXFN function.
3. Loads and relocates of one or more assembly code segments from a disk volume through the System Loader.

Invokable module description

An invokable module is a relocatable assembly-language load file produced by the Apple IIGS APW Assembler and Linker or its equivalent. The file must conform to the definition of a ProDOS OMF load file, as required by the System Loader. The definition of a load file, a type of OMF file, is included in the *Apple IIGS Programmers Workshop Reference* manual. The System Loader is described in the *Apple IIGS ProDOS 16 Reference* manual.

A single invokable module can contain from 1 to 255 entry points. These are specified by the entries in the separate DICTIONARY data segment you must create with the APW Assembler. The name you give to each entry point is the symbolic name that will be used by PERFORM and EXFN to access the assembler routine.

An invokable module, like all load files, can not have any unresolved symbolic references. The first segment in the load file, number 001, must be a static code segment, and all entries in the DICTIONARY segment are, by definition, entry points relative to the origin of segment 001.

❖ *Technical Note:* An invokable module could have a segment jump table and additional dynamic code segments, but handling the memory management requirements of dynamic segments is the responsibility of those who understand how to write such applications. IIGS BASIC maintains a number of locked Memory Manager segments, and there is no mechanism for an external routine to unlock them or cause them to move independent of the interpreter—you are on your own when you use dynamic segments as part of an invokable module.

The pointer addresses created by IIGS BASIC are passed to invoked modules as four bytes on the stack, with the most significant byte a 0. IIGS BASIC always calls an invokable module with the direct-page register set for the interpreter direct page and the data bank register (DBR) for the interpreter static data bank. Both of these registers must be preserved by an invokable module.

Initially, the BASIC DBR will probably match its program bank register (PBR), but they may be different in future releases of the interpreter. If an invokable module must reference the interpreter code bank, it resides in a byte of the BASIC zero page. The data bank address should always be determined from the DBR ; do not assume that it matches the PBR.

The word count table below shows stack space used for the various types of parameters that BASIC can pass.

Table I-1

Parameter description	Type char	Word count	Byte count
Structure element	!	1	2
Regular integer	%	1	2
Double integer	@	2	4
Long integer	&	4	8
Single precision real	none	2	4
Double precision real	#	4	8
VARPTR address	NA	2	4
String address	$	2	4

BASIC does not allow variable length parameter lists. Thus, parameter passing requires a fixed number, order, and type of arguments. An incorrect number or type of argument will generate an error message.

DICTIONARY segment definition

The data generated within the DICTIONARY segment, for each entry point in the code segment, defines for IIgs BASIC all the information required to call the entry point for the PERFORM statement or the EXFN function reference. The DICTIONARY consists of a short header, followed by a table of interface definitions, one per entry point. These table entries could be generated with a macro (if you want to tackle writing one to do the job).

The following is an example of how to code an invokable module and generate the interface definitions of the DICTIONARY segment:

```
            KEEP        INVOKE
            65816       ON

ORIGIN      START       MYSWEETCODE
            RTL
            END

;  equates, MLOADs, and other such stuff goes here

ENTRY_PT1   PRIVATE     MYSWEETCODE
            USING       …
            SetMode16
            PLA
            …
            END

ENTRY_PT2   PRIVATE     MYSWEETCODE
            USING       …
            SetMode16
            PLX
            PLA
            …
            END

;           and so on

IDEF        PRIVDATA    DICTIONARY ;interface definition table
            DC          I1'02'     ;two entry points
            DC          I1'01'     ;dictionary format version 01
            DC          I2'IDEND-E1' ; total length of dictionary table

E1          DC          I2'E1end-E1 ; length of this entry
E1_ELen     DC          I1'0'          ;filled in by IIgs BASIC
E1_Entry    DC          I2'ENTRY_PT1-ORIGIN' ;offset to entry point
E1_Segnum   DC          I2'0001'       ;always = 0001
E1_RSize    DC          I1'00'         ;result stack size in words
E1_ERRMd    DC          I1'02'         ;error-handling mode
E1_TYP      DC          I1'01'         ;type of entry point: PROC=01
E1_FNTyp    DC          I1'$81'        ;type of function result
```

```
E1_PCnt     DC          I1'02'          ;input param count
E1_POFS     DC          I1'E1_PT1-E1';offset to E1_PT1
E1_NLen     DC          I1'L:E1_Name';length of E1_NAME
E1_Name     DC          C'APROC1'   ;    name of the procedure
E1_PT1      DC          I1'02'      ;parameter type for input argument #1
E1_PT2      DC          I2'03'      ;parameter type for input argument #2
E1end       ANOP

E2          DC          I2'E2end-E2 ; length of this entry
E2_ELen     DC          I1'0'           ;filled in by IIGS BASIC
E2_Entry    DC          I2'ENTRY_PT2-ORIGIN' ;offset to entry point
E2_Segnum   DC          I2'0001'        ;always = 0001
E2_RSize    DC          I1'02'          ;result stack size in words
E2_ERRMd    DC          I1'01'          ;error-handling mode
E2_TYP      DC          I1'02'          ;type of entry point: PROC=01
E2_FNTyp    DC          I1'02'          ;type of function result
E2_PCnt     DC          I1'02'          ;input parameter count
E2_POFS     DC          I1'E2_PT1-E2';offset to E2_PT1
E2_NLen     DC          I1'L:E2_Name';length of E2_Name
E2_Name     DC          C'AFUNCN2'  ;    name of the procedure
E2_PT1      DC          I1'02'      ;parameter type for input argument #1
E2_PT2      DC          I2'03'      ;parameter type for input argument #2
E2end       ANOP

IDEND       ANOP
            END
```

The values to select for the various options of each field are defined in the next section. The format of the DICTIONARY segment is used as is (except for the header) at run-time in the IIgs BASIC INVOKE IDT. The first field is copied to the second and then is replaced with a linked list pointer. Generating invalid data in the DICTIONARY segment will cause the message

```
?INVALID DATA ERROR
```

to appears when a PERFORM or EXFN referencing the invalid entry is executed.

The interface definition entry

The following describes the interface definition entry format that must be generated in the DICTIONARY segment of an invokable module.

Table I-2

Field name	Size	Description
ID_NXT	Word	Length of the entire entry (including ID_NXT)
ID_ELEN	Byte	Zero
ID_ENTRY	Word	Entry point offset (in memory) within the static code segment segment 001 of an invoked load file.

ID_Segnum	Word	Load segment number of entry point; always 0001
ID_RSIZ	Byte	Result stack size in words (0 through 16)
ID_ERRM	Byte	Error-handling mode
	$00	N = No error result is returned
	$02	E = Carry set := X-register (8-bit) is BASIC error #
	$04	P = Carry set := A-register (8-bit) is ProDOS error #
	$06	T = Carry set := C-register (16-bit) is tool set error #
		The following error-mode values may have $80 bit on to indicate EXFN_ may call the procedure as if it were a function.
	$08	C = Carry set := return carry as an integer result
	$0A	X = Carry set := return X-register (8-bit) as an integer result
	$0C	A = Carry set := return A-register (8-bit) as an integer result
	$0E	B = Carry set := return C-register (16-bit) as an integer result
ID_TYP	Byte	Type of entry point in the invoked module
	$01	Procedure (ID_RSIZ usually zero)
	$02	Function (ID_RSIZ must not be zero)
ID_FNTYP	Byte	Type of function result
	$00	Indicates multiple results or no results on stack; for multiple results, ID_TYP must indicate a PROC; user must use R.STACK functions to get the results
	$01	I = Integer result (a word)
	$02	L = Double integer (a long)
	$02	A = Address or pointer (24 least significant of 32 bits)
	$03	C = Comp integer or BASIC long integer
	$04	F = SANE single-precision real result
	$05	D = SANE double-precision real result
	$06	X = SANE extended-precision real result
	$07	T = String descriptor (convert 3-byte pointer + length byte)
	$0F	P = Pointer to a counted string result (convert it)
	$17	Z = Pointer to a C-string (0-ended) result (convert it)
ID_PCnt	Byte	Input parameter count (0 to 10)
ID_POffset	Byte	Offset from ID_NXT to PT1 item below
ID_NLen	Byte	Length of ID_NAME item (1 to 20 characters)
ID_NAME	1 to 20	Name of the procedure or function (A-Z, 0-9 only)
ID_PT1	Byte	Parameter type for input argument 1

ID_PT2..10	Bytes	Parameter type byte(s) for input argements 2 through 10
	$01	I = Integer argument (a word)
	$02	L = Double-integer argument(a long)
	$02	A = Address argument (24 least significant of 32 bits)
	$03	C = Comp integer argument (BASIC Long Integer)
	$04	F = SANE single-precision real argument
	$05	D = SANE double-precision real argument
	$06	X = SANE extended-precision real argument
	$07	S = Address of BASIC string descriptor argument
	$87	P = Pointer to a counted-string argument created from a BASIC string (descriptor address)

Parameter description and conventions

The specific order and type of parameters chosen for an invokable module are up to you. You can also decide either to pass a parameter by address or by value. The parameters are processed from left to right and pushed on the stack in that order; thus, the last parameter in the argument list will be the first to be pulled off the top of the stack.

Parameters are passed by address by using the VARPTR or VARPTR$ functions. String parameters are discussed a little later.

The PERFORM and EXFN interface (but not the tool set CALL interface) passes all the input arguments on the CPU stack *above* the RTL address for BASIC. Any result space for functions (always zeroed) is *below* the RTL address on the stack, so an invokable module does **not** have to remove the return address to pop the input arguments, or move the return address before executing an RTL.

The following is the order of items on the stack when a procedure or function is entered are as follows (where ss is the page address of the stack pointer):

Table I-3

Stack contents	Stack address	Example value
(Prior contents)	$00*ss*DF	
Space for result	$00*ss*DB	$00000000
RTL address	$00*ss*D7	$0329FE
Argument 1	$00*ss*D5	$0435
Argument 2	$00*ss*D1	$00000211
. . .		
Argument *n*	$00*ss*CF	$0001
	$00*ss*CE	¨ top of Stack

The example stack addresses shown above are for the low byte of the example values, which are shown as assembler constants, not in their byte order when stacked. For example, the example RTL address refers to the $FE byte, while the $03 byte would be in location $00*ss*D9.

❖ *Note:* All parameters that are passed by address must be accessed via the 65816 long indirect addressing mode. This is the only means of access that will work because the assembly code of an executing invoked module will probably not be in the same bank as the variable storage tables.

Arguments are pushed on the stack high (or most significant) byte or word(s) first; therefore, the invoked code must pull parameters off the stack low (or least significant) byte or word(s) first. This is shown below (in bytes) for some of the parameter types. In the table, MSB means most significant byte and LSB stands for least significant byte.

Table I-4

Real	Integer%	Address	Long Integer&	Stack Address Example	
			MSB	$ssE9	
			.	$ssE8	
			.	$ssE7	
			.	$ssE6	
MSB		zero	.	$ssE5	
.		BankByte	.	$ssE4	
.	MSB	PagByte	.	$ssE3	
LSB	LSB	LowByte	LSB	$ssE2	
				$ssE1	" Top of stack

Double integers require 4 bytes (like the address example) and double-precision reals require 8 bytes (like LongInteger&).

String parameters

String parameters are only passed by address. There are two ways to pass a string, both of which push an address on the stack. First, you can select the type of an input string parameter so that the address of a BASIC string descriptor is passed on the stack. This method is the fastest because it passes the address normally created by BASIC for a string. However, the address of a string descriptor only indirectly implies the address of the string's data; how to use string descriptors is described in the next section.

A second method, called the P-string method, is easier to use but slower. This method converts a BASIC string descriptor and its data into a counted string, also called a Pascal or P-string. The address of the count byte is then pushed on the stack as the argument. The P-string option is slower because BASIC must copy the string data and move the length byte from the descriptor to the string data pool.

Trying to create new variables, particularly strings, or to change the length of a string variable involves the entire user data segment of BASIC. This is necessarily so since either of these actions could require *garbage collection,* or compression of the string literal pool to recover free memory for the new string.

Warning

IIgs BASIC provides an interface that allows access to some of the interpreter's internal subroutines that process strings and perform other functions. This work-saving interface, described in a later section, is the only interface to the internals of the BASIC interpreter that will be supported in future releases of Apple IIgs BASIC.

String internals

You must understand the difference between a *pointer to a string descriptor*, a *string descriptor* and a *pointer to the string data* to sucessfully write an invokable module that directly stores strings into BASIC's variable tables.

However, EXFN can be used to perform this complex task by creating a string function and simply using the external function in a string expression or LET statement. You can skip this technical discussion if your invokable modules will just return string results via string functions. The internal subroutines described later, such a PTRGET, normally return a pointer to a string descriptor.

BASIC uses a 3-byte descriptor as the value of a string. Thus, the address passed by BASIC for a string argument is nominally the address of this 3-byte descriptor, not a pointer to the string's actual character data.

A string descriptor consists of a length byte, followed by a relative offset. The length is an unsigned value, with 0 indicating that the string is null. The next 2 bytes are a relative offset to the beginning of the string data. The offset is an unsigned number that is subtracted from the highest address for the string data pool to locate the beginning of the actual ASCII characters. This offset is invalid when the length is 0, so the length must be checked before using the relative offset.

More than just the content of a string is stored in the string data pool. It also contains a 3-byte descriptor that points back to the string's descriptor in the variable tables. This *back pointer* consists of a type byte and an unsigned relative offset from the beginning of the simple or local variable table or the array table. The type byte normally specifies to which of the three tables the offset applies.

Because the descriptor and the back pointer have 16-bit relative offsets, all the string descriptors must be placed within the first 64K of either the simple or array variable tables, and the string data pool can't be larger than 64K.

There are two types of BASIC string descriptors: temporary and variable. A temporary string descriptor is created for the result of a string expression or string sub-expression. A temporary descriptor is 3 bytes in the temporary string descriptor stack, named TMPDSCSTK, located in the BASIC interpreter zero page. String variable descriptors are the value portion of the string variable or array element entry in the BASIC storage tables.

The actual characters of a string variable are stored separately from (and located by using) the string descriptor. Thus, a pointer to a string descriptor is like a Memory Manager handle; it must be dereferenced into a pointer before the string data can be referenced.

A string descriptor cannot be dereferenced like a handle to directly create a pointer to the string's data. It first must be converted into a pointer by subtracting the relative offset from a pointer, named LITEND, in the interpreter zero page. An internal routine, named NOTNOW, performs this conversion, using certain zero-page work pointers.

If you want to assign some ASCII data to a string variable, you must copy the ASCII data into BASIC's string literal pool and build a temporary descriptor for that data. Then you can copy the new descriptor into the descriptor of a string variable and update the back pointer in the string data pool to point to the variable's descriptor.

Before replacing the existing descriptor for a string variable with the new one, the old string data must be deallocated. Deallocation involves changing the string literal pool back pointer to indicate the size of the free memory space that is being deallocated.

BASIC's STRCP and STS2M routines, perform these two tasks for an invokable module. STRCP copies the ASCII data into the string data pool, creates the associated null back pointer, and builds a temporary string descriptor for the data.

If the DOSUBEXPR routine is used to evaluate a string expression, the string result is returned as a pointer to a temporary descriptor, as if STRCP was used to create it. Temporary descriptors are stored in a zero-page stack. Temporary descriptors are exactly like variable descriptors, and they can also be converted with NOTNOW if you need to reference the ASCII string data.

STS2M (which stands for store string to memory) assigns a temporary descriptor to a variable descriptor deallocates the old one, and fills in the null back pointer.

Using strings in BASIC is complex. We recommend that you clarify your understanding of how it all works by explaining the process to another programmer.

Interpreter internals

IIGS BASIC is a tokenizing interpreter that compresses, upon program entry, keywords, line numbers and integer constants into 1-, 2-, 3-, 4-, and 5-byte tokens. To execute a program, BASIC scans the tokenized form of a statement byte by byte using its zero-page program pointer, named TXTPTR.

All statements (except LET) begin with a verb token and the verb's routine is called by indexing into an address table with the token. Each statement routine performs additional syntax checking each time a statement is executed. TXTPTR's position determines what line and statement of a program is executing.

Some of the routines that can be called by the DISPATCH interface will modify the TXTPTR that is normally pointing to the end of the PERFORM or EXFN parameter list. Because the value of TXTPTR is the means of continuing with the program after returning from an invoked procedure or function, TXTPTR must be preserved if routines that modify it are called by the invoked code.

An example of this is the GETNAME routine for creating the pathname for opening a file. GETNAME must have the TXTPTR modified to point to the pathname to be opened. If the existing TXTPTR (all three bytes, remember) is not saved and restored, BASIC will attempt to continue executing your program from the data at the end of the pathname, where GETNAME left TXTPTR, when you use a RTL instruction to return to BASIC.

All the subroutines available through the DISPATCH interface are called in 8-bit native mode and return in that mode. Most of the interpreter is coded in 8-bit native mode, with liberal mode switching to full 16-bit and mixed 8- and 16-bit modes. The standard calling convention for almost all subroutines is that they are called and returned in 8-bit native mode, with a few exceptions.

IIGS BASIC implements the TXTPTR as a normal long-indirect pointer (3 bytes) which resides in BASIC's zero page. CHRGET and CHRGOT are subroutines for fetching program text. These routines are useful for syntaxing tokenized ASCII data. CHRGET gets the next sequential nonspace character or token from the data at TXTPTR. Upon return from CHRGET, the A register contains the character or token, and the CPU status flags are set as follows:

- Z flag is set (BEQ) if the character is a colon ($3A) or a line terminator ($00). It is reset (BNE) otherwise.
- Carry is clear (BCC) if the character is an ASCII digit. It is set (BCS) otherwise.
- N flag is set (BMI) if the byte is a token with a value more than $B9 and carry is set ($BA and up because CMP #':' is equivalent to $BA ...-$3A = $80 ...).

The interface has been specifically designed to maintain the separation of code within the interpreter and the invokable modules, while still allowing access to the interpreter. This should allow the interpreter to be modified and upgraded in the future without changing existing invokable modules. The call interface consists of a routine number and an entry point within the interpreter's zero page.

No fixed-address vector is available to construct an assembly-time interface for calling BASIC's internal routines. All of BASIC itself is relocatable, even its zero page. Nonetheless, an easy-to-use interface has been devised so an invokable module can call BASIC without resorting to self-modifying code.

Since invokable modules are called from BASIC, the direct page register will contain the bank zero address of the interpreter's zero page. During cold start, the interpreter constructs a JMP long instruction starting at byte 1 of its zero page. This JMP instruction may be easily executed through an elegant three-instruction sequence.

To call the interpreter, an invokable module will JSL to the following three instructions:

```
GOBASIC     PEA     $ff00
            PHD
            RTL
```

The operand of the PEA instruction must consist of a routine number, shown above as $ff, with a low byte of zero. Effectively, what these instructions accomplish is a self-relocating JMP long to $00*dd*00+1, where *dd* is the page address of the interpreter zero page from the DP-register, while leaving the routine number on the top of the stack. This sequence will always work, no matter what zero page was assigned to the interpreter by the System Loader (it would even work if the zero page isn't page aligned).

A separate set of these three instructions is required for each internal routine to be called because they contain the routine's number. Note that this interface has the important advantage of not modifying any of the processor registers, which are often used as inputs to BASIC internal routines.

The JMP long in zero page then gives control to a dispatcher in BASIC, which removes the routine number from the stack and calls the appropriate internal routine (while preserving the input registers). The dispatcher effectively does a JSR to the BASIC routine, which returns via an RTS instruction. Then the dispatcher returns to the calling external module via an RTL instruction. This interface returns all the registers and the processor status unchanged for use by the invokable module.

Most, if not all, of the routines that can be used within the interpreter can exit through the error paths of the interpreter and not return to the caller. A vector exists to allow an invokable module to regain control when this happens. This requires that the BASIC program calling the invoked module prepare the vector beforehand to properly handle such errors.

ProDOS calls can be made directly to ProDOS from invoked procedures or functions. Doing so can be useful, and very dangerous, too. IIGS BASIC may have files open, memory allocated, events armed, and so on, and a change in the assumed state of these things will not be reflected in the status information maintained by BASIC. When BASIC acts on its incorrect information, a system hang or crash could result. This does not preclude making ProDOS calls, but you should carefully select what your invokable module will do directly with the operating system.

Of particular concern is the limit on the number of open files allowed under ProDOS 16 version 1.2; an invokable module should never tie up the last available ProDOS FCB because BASIC assumes that one will always remain available for the CAT, CATALOG, DIR, and TYPE statements. The COPY command requires that two FCBs remain available.

External routine summary

This table below describes the internal routines currently accessible through the IIGS BASIC DISPATCH interface. References to specific pointers and work areas are shown in all capital letters and are defined in a later section.

Table I-5

Name	Number	Description
CHRGET	0	Increments TXTPTR and returns the next nonspace program byte.
CHRGOT	2	Returns the current program byte (other than a space)
---(BETA2 rel)	4	Evaluates an expression; returns type and result in XACC (a floating-point accumulator)
PTRGET	6	Finds, in the variable tables, the variable or array element at TXTPTR.
ERRDIR	8	Returns if a program is running; otherwise an else illegal direct error occurs.
LINLBLGET	10	Converts the line number or label at TXTPTR into LINNUM/LINLABL
LINNUMGET	12	Converts the line number at TXTPTR into LINNUM as binary integer
GOTO	14	Returns TXTPTR at the line given by line number label at TXTPTR
GOTOB	16	Returns TXTPTR at the line in LINNUM/LINLABL if existent; otherwise an error occurs
GETADRS	18	Convert numeric expression to double integer and put in FORPNT as address
FINDLIN0	20	Search forward from Y-X-A in program for line number >= linenum used elsewhere in book

FINDLIN	22	Same as FINDLIN0 (routine number 20), but search starts at the beginning of the program
NOTNOW	24	Given a pointer to a string descriptor, returns string data pointer and the length in A register
ERROR	26	Raises the BASIC error condition given in the X never returns
SERROR	28	Raises (if possible) the BASIC error matching the ProDOS error in the A register.
UMSHRINK	30	Requests BASIC to free up A register pages of data segment memory to ProDOS
UMEXPAND	32	Inverse of SCRUNCH Expand memory for data segment by A register pages
FRECNOW	34	Frees up string descriptor and string space at pointer in XACC
GETNAME	36	Converts ASCII data at TXTPTR into pathname in NAMBUF
FOPEN_AX	38	Opens the filename in NAMBUF, with access in the A register and file type in X
AY2XINT	40	Creates an integer of A-Y and returns it as an integer in XACC
POSINT	42	Checks XACC for positive and converts XACC to an integer in A-Y
CVTTXT2X	44	Converts ASCII data at TXTPTR into numeric value in XACC
DATAN	46	Moves TXTPTR to the end of the current tokenized statement
STRCP	48	Copies your string data at STRNG1 into string pool and builds a temporary string descriptor pointing to it
STS2M	50	Stores a string (pointer in FACPTR) to a string variable at FORPNT
STX2V	52	Stores a value in XACC into a variable of type VTYPE at FORPNT.
CONV2STR	54	Converts XACC into a string and returns descriptor pointer in FACPTR

Functions and their results

As discussed previously external functions in IIGS BASIC operate through the EXFN statement. EXFN is used to return any type of numeric or string result. All numeric results are returned on the stack by value. String results are returned by reference; that is by returning the address of the result on the stack. Strings can be returned in the following three ways:

- A combination of a 3-byte address and a 1-byte length can be returned on the stack. This is called the *descriptor result mode* because of its similarity to BASIC's string descriptors. The least significant 3 bytes of the 4-byte result are the address of the string data, and the most significant byte is the length.
- A pointer to a P-string, or counted string can be returned on the stack.
- A pointer to a C-string, or zero ended string, can be returned on the stack.

For all three cases, BASIC converts the resulting pointer and data into a BASIC string descriptor and copies the string's data into the string literal pool. This interface makes it very easy to construct string functions with invokable modules.

The EXFN results must be stored on the stack in the same manner as they are passed, that is, with (the least significant byte)on top of the stack just below the return address. All integer results are signed two's complement integers, with the sign bit taken as bit 7 (MSBit) of the highest byte.

The space for a function result is preallocated on the stack, below the return address (3 bytes) by the PERFORM/EXFN interface. The 65816 processor has a stack-relative address mode that can be used to store data in the preallocated result space without having to remove the return address, push the results, and replace the return address.

Real results are formatted according to the definitions of the 65816 SANE math engine. Refer to the *Apple Numerics manual* and the SANE chapter of the *Apple IIGS Toolbox Reference* manual for details. BASIC can accept single, double; or extended-precision results from an external function *on the stack*. Numeric results can not be returned by reference; that is, by returning an address of the numeric data.

Interfacing with the BASIC interpreter

Invokable modules used with IIGS BASIC commonly need to access some internal subroutines within the interpreter. A special interface, called DISPATCH, is provided for this purpose; it is the *only* interface that will be supported in future releases of the BASIC interpreter. Use of any other interface may become invalid upon the next release of IIGS BASIC.

CHRGOT returns the current character at TXTPTR if it is not a space ($20), otherwise, it returns the next nonspace character. Thus does the same things as CHRGET, except that the TXTPTR is not incremented before fetching the character. CHRGOT is normally only used after a call to CHRGET. When this is the case, CHRGOT will not change the TXTPTR because CHRGET will never leave TXTPTR pointing to a space.

Almost all the zero-page pointers defined later are 3-byte long-indirect pointers, so beware of INC and STA instructions in 16-bit mode. A few of them pointers are 4-byte long-indirect pointers. Before you begin writing invokable modules that use these routines, you must have a complete understanding of the 65816 long-indirect addressing mode.

BASIC defines two floating point accumulators, known as the XACC and YACC. These two 10 byte areas are in BASIC's static data segment, and they contain the values for integer and real numbers. Each accumulator has a matching packet of status information, in zero page, that defines the type, class, size, and numeric status of the operand currently stored in the accumulator. The packet includes a 4-byte pointer that can be used to reference the accumulator's value. The packet also contains a string pointer that always points to a string descriptor, either a temporary descriptor or one in the variable/array storage tables.

Pointers to string descriptors are left in FACPTR, FACPTR+1 and FACPTR+2 (or FACPTRB). Routine 48, STRCP, creates a temporary string descriptor for your string data (possibly read from a storage volume). These temporary descriptors must be released with FRECNOW before returning to BASIC if they are used by an invoked module. The temporary descriptor stack is located in zero page, and so an address in FACPTR, left by STRCP, will be of the form $00*ddxx*, where *dd* is the DP register and *xx* is the address of the temporary descriptor in zero page.

IIGS BASIC zero-page equates

Table I-6

Address	Name	Description
$00	MYBANK	Contains bank address of BASIC's code segment
$01-04	DISPATCH	Entry point for BASIC internal routine dispatcher
$05	ZPAGREG	Bank zero page address of BASIC's zero page
$07,08	BBUSERID	User ID assigned to BASIC code segment
$0F	VTYPE	Output variable type flag from PTRGET
$10	VCLAS	Output variable class flag from PTRGET
$11	AUTODIM	Must be $00 for PTRGET and ANYSUBEXPR calls
$1B	SVFILNO	Save temp for file (FCB) # of current operation
$1C-1E	DELTA	Length temp for memory management subroutines pointer adjustments

$1F-21	VARNAM	Temps used by PTRGET and its subroutines
$22-24	VARPNT	Output pointer from PTRGET and others
$25-27	VARTXT	TXTPTR save temp used all over the place
$28-2A	DEFPNT	Pointer used in DEF PROC/FN, PROC, FN processing
$30,31	LINNUM	Temp for input and output of line numbers
$40-43	INDEX	General-purpose pointer used by many routines
$44-46	INDEX2	General-purpose pointer used by many routines
$47-4A	WORKPTR	General-purpose pointer used by memory management routines
$4B-4D	TXTTAB	Pointer to lblfld byte of first line in program segment
$4E-50	TXTPTR	Current character (or token) pointer in program
$51-53	LCLEND	Temporary end of new local variable table during argument processing
$54-56	LCLTAB	Pointer to current local simple variable table
$57-59	VAREND	Pointer to end of (global) simple variable table
$5A-5C	VARTAB	Pointer to beginning of (global) simple variable table
note1	PROCTAB	Pointer to procedure/function name table
$5D-5F	ARYTAB	Pointer to beginning of array table
$60-62	STREND	Pointer to end of local variable table
$63-65	FRESPC	Work pointer used in allocating next string literal
$66-68	FRETOP	Pointer to lowest address allocated in string literal pool
$69-6B	LITEND	Pointer to end of string literal pool + 1; same as INVTAB
"	INVTAB	Pointer to begin of invoked segment record table
$6C-6E	USREND	Pointer to end of data + 1 in user data segment
$6F-72	PROCTAB	(temporarily here out of the way for BETA 1 release)
$72-74	LIBTAB	Pointer to tool set library dictionary (linked list of tables)
$75,76	CURLIN	Current line number in binary (low byte first)
$7C,7D	DATLIN	Line number of current DATA statement (for errors)
$7E-80	DATPTR	Pointer to program DATA statement text
$84-86	FORPNT	Output data from GETADR routine (same as LINNUM)
$87-89	DSCPNT	Temporary pointer to string descriptor in string routines
$8A	FOUR6	Temporary work byte used during PTRGET and elsewhere
$8B-8D	HIGHDS	Address of highest destination byte for move routines; also used during PTRGET for variable and array lookup
$8E-90	NDXPTR	Record buffer data pointer for disk I/O operations
$91-93	HIGHTR	Address of highest source byte to move for move routines
$94-96	ENDPTR	End of table ptr in PTRGET search + garbage collection temporary
$97-9A	LOWTR	Block move lowest source address; FINDLIN result
$9B-9E	LONGPTR	Handle dereference pointer and memory management routines
$9F-A2	X.PTR	Pointer to XACC (never changes; don't touch)
$A3	XSDTYP	XACC Sane data type byte = 03,03,04,05,02,01,00,--
$A4	XTYPE	XACC type# = 00,01,02,03,04,05,06,07
$A5	XCLAS	XACC class# = $40,$40,$40,$40,00,00,00,$80

$A6,A7	XBYTS	XACC byte count = 01,02,04,08,04,08,$0A,n.a.
$A6-A8	FACPTR	Pointer to string descriptor for XTYPE = 07 only
$A9	X.STS	Zero/nonzero status for numeric XACC (XTYPE <> 7)
$AA	X.SGN	$00/$80 sign status for numeric XACC (XTYPE <> 7)
$AB-AE	DSCTMP	Temporary work string descriptor length +3 byte address
$AF-B2	Y.PTR	Pointer to YACC (never changes)
$B3	YSDTYP	YACC SANE data type byte = 03,03,04,05,02,01,00,--
$B4	YTYPE	YACC type# = 00,01,02,03,04,05,06,07
$B5	YCLAS	YACC class# = $40,$40,$40,$40,00,00,00,$80
$B6,B7	YBYTS	YACC byte count = 01,02,04,08,04,08,$0A,n.a.
$B6-B8	ARGPTR	Pointer to string descriptor for YTYPE = 07 only
$B9	Y.STS	Zero/nonzero status for numeric YACC (YTYPE <> 7)
$BA	Y.SGN	$00/$80 sign status for numeric YACC (YTYPE <> 7)
$BB-BD	STRNG1	Input pointer for STRCP
$D4	REFTBL	Output from PTRGET, input for STX2V
$E7	IMODE	$0X = deferred mode, $8X = immediate mode
$F5,F6	SWCHGO	Temps used only by DISPATCH entry point (above)
$F8-FF	unused	Available for invokable module temporary work pointers

❖ *Note 1:* The pointer named PROCTAB is going to move from its current temporary location to the position indicated by the 'note 1' reference. The pointer from ARYTAB thru USREND will all move up three bytes when PROCTAB is implemented in the BETA2 release.

These zero page equates are not likely to need to change after final release, but ARE GOING TO CHANGE IN THE **BETA 2** release.

Warning

Be sure that all references to zero-page locations are defined with equates so that invokable modules can easily be reassembled when changes are required later.

Internal subroutine descriptions

CHRGET (routine number 00)

Description: CHRGET gets the next sequential nonspace character or token from the program text by incrementing the TXTPTR and then fetching the next byte from program.

On entry: The routine must be entered in 8-bit native mode, as must all the other routines described here.

On return: Upon return from CHRGET, the A register contains the character or token and the CPU status flags are set as follows:

- Z flag is set (BEQ) if the character is a colon ($3A) or the end-of-line terminator ($00). Z flag is reset (BNE) otherwise.
- Carry is clear (BCC) if the character is an ASCII digit . It is set (BCS) otherwise.
- N flag is set (BMI) if the byte is a token with a value more than $B9 and carry is set ($BA and up because CMP #':' is equivalent to $BA ...-$3A = $80 ...).

Error exits: None.

CHRGOT (routine number 02)

Description: CHRGOT gets the current nonspace character or token from the program text by fetching the byte at the TXTPTR from program.

On entry: The routine must be entered in 8-bit native mode.

On return: Upon return from CHRGET, the A register contains the character or token, and the CPU status flags are set the same as CHRGET above.

Error exits: None.

--- (routine number 04)

Description: Routine # 04 will be changed and documented in the BETA 2 release.

PTRGET (routine number 06

Description: PTRGET searches the variable storage tables for a variable or array element and returns the address of a variable or element. The variable name pointed to by the current TXTPTR is found or created in memory and a pointer to its value is placed in VARPNT. This routine calls DOSUBEXPR to evaluate array subscripts. DOSUBEXPR may call PTRGET to locate variables (note the recursion).

PTRGET always searches the current local variable table, if one exists If it does not find the variable there, it searches the global simple variable table. If the variable name is followed by subscripts enclosed in parentheses, PTRGET searches the array variable table after searching the local one.

On entry: TXTPTR points to the variable to be referenced.

On return: VARPNT points to the variable value.

X, Y, and A registers contain the pointer to the variable value.

B register contains the value of REFTBL below.

VTYPE and VCLAS determine the variable's type, as shown in the table below.

REFTBL is 00 if a simple variable was found, $40 if a local simple variable was found, or $80 if an array element was found.

Error exits: Array subscripts can be expressions, So most of the DOSUBEXPR error exits could be taken.

Table X-X
Tabletitle

Variable type	VTYPE	VCLAS	XACC value organization
Structure element	00	$40	XACC (XACC+1 = zero)
Single integer	01	$41	XACC and XACC+1
Double integer	02	$42	XACC ... XACC+3
Long integer	03	$43	XACC ... FAC+7
Single real	04	$04	XACC ... XACC+3
Double real	05	$05	XACC ... XACC+7
Extended real	06	$06	XACC ... XACC+9
String	07	$87	Pointer in FACPTR

The value of REFTBL indicates which table contains the address returned by PTRGET. A structure element is only returned for an array reference, and the extended real type is never returned by PTRGET because extended real is not an explicit BASIC variable type.

ERRDIR (routine number 08)

Description: ERRDIR Validates that a program is executing and returns.

On entry: None. Destroys the X register.

On return: In deferred mode, that is, executing a BASIC program.

Error exits: If in direct mode, the

```
ILLEGAL DIRECT ERROR
```

message is sent to the current output stream device, normally the console. Control then returns to direct mode.

LINLBLGET (routine number 10)

Description: LINLLGET converts a line number pointed to by TXTPTR into a 16-bit integer in LINNUM. Alternately, LINLBLGET copies a line label into a fixed-address work buffer named LINLABL, from the program text. The entry requirements are met by loading the A-reg via CHRGET or CHRGOT. LINGET will accept unsigned integers in the range of 1 through 65279. A line number may be tokenized as a binary number, as ASCII digits, or as a line label.

There must be a nondigit or nonlabel character following the valid digits or label characters. LINGET will stop on the first nondigit or nonlabel character it encounters and return the accumulated result. Leading zeros will be ignored in line numbers. When LINLBLGET encounters a label, LINNUM is returned with the value zero. When a LINNUM is returned, LINLABL and LINLABL+1 will contain zeros.

On entry: TXTPTR points to the first byte of the line number or to the first character of a label (a letter A-Z, a-z) A register contains first character of line number or label. Carry must be clear for a digit in the A register. Carry must be set for either a label or tokenized line number. N flag must be set for a tokenized line number.

On return: LINNUM, LINNUM+1 is the 16-bit equivalent, low-high.
TXTPTR will point to the terminating character.
Uses location INDEX for work temp.

Error exits: Illegal line number/label error will result if a zero or a number higher than 65279 is input.

LINNUMGET (routine number 12)

Description: LINNUMGET converts a line number pointed to by TXTPTR into a 16-bit integer in LINNUM. The entry requirements are met by loading the A register through CHRGET or CHRGOT. LINGET will accept unsigned integers in the range of 1 through 65279. A line number may be tokenized as a binary number or be input as ASCII digits.

There must be a nondigit following the valid ASCII digits. LINGET will stop on the first nondigit and return the accumulated result. Leading zeros will be ignored in line numbers. When a LINNUM is returned, LINLABL and LINLABL+1 will be unchanged.

On entry: TXTPTR points to the first byte or ASCII digit of the line number.

A register contains the first byte of the line number.

Carry must be clear for a digit in the A register.

arry must be set for a tokenized line number.

N flag must be set for a tokenized line number.

On return: LINNUM, LINNUM+1 is the 16-bit equivalent, low-high. TXTPTR will point to the terminating character or the first character following the tokenized form of a line number.

Uses location INDEX for work temp.

Error exits: Illegal line number/label error will result if a zero or a number higher than 65279 is input.

GOTO (routine number 14)

Description: GOTO searches the resident BASIC program for the line number or label at the current TXTPTR address. The line number or label is setup for GOTOB by calling LINLBLGET (described above). The line number or label must be followed by an end-of-statement $00 or ':'.

On entry: Status (P register) and A register set by calling CHRGOT or CHRGET.

On return: Sets TXTPTR to point to the end-of-line token ($00) preceeding the desired line. IMODE, a zero-page variable, will have had its $80-bit reset.

Error exits: The undefined statement error exit is taken if the LINNUM or LINLABL is nonexistent. Illegal line number/label error occurs if a line number of zero or more than 65279 is found.

GOTOB (routine number 16)

Description: GOTOB searches the resident BASIC program for line number given by LINNUM, LINNUM+1, or the label in LINLABL. Searches forward in the program if CURLIN is less than LINNUM; otherwise, it searches the program from the beginning. When a label is used, the entire program is always searched.

On entry: LINNUM is the line number to locate, or LINLABL is the label to locate, and LINNUM is zero.

On return: Sets TXTPTR to point to the end-of-line token (0) preceding the desired line.

Error exits: The undefined statement error exit is taken if LINNUM or LINLABL is nonexistent.

GETADRS (routine number 18)

Description: GETADRS converts the numeric expression beginning at TXTPTR into the XACC accumulator and then converts XACC into a double-integer format number. The double integer is then checked to be sure that the value is in the range of 0 through 2 24-1.

On entry: TXTPTR points to the first character of the tokenized numeric expression.

On return: FORPNT contains the address from the numeric expression. XACC contents destroyed.

Error exits: An illegal quantity error exit will occur if the numeric value is outside the range of 0 through 2 24-1.

FINDLIN0 (routine number 20)

Description: FINDLIN0 searches through the BASIC program for the line number given in LINNUM or the label given by LINLABL. LOWTR is set to point at the label field length byte of the first line encountered with a line number greater than or equal to LINNUM when searching for a line number, and points to the line or end-of-program when searching for a label.

On entry: LINNUM contains the line number to search for in the program, or LINLABL contains the label to search for. The Y, X, and A registers contain the bank-high-low beginning search address pointer and must point to the label-field length byte of an existing line.

On return: LOWTR is a pointer to the first line whose number is >= LINNUM. Carry is set if a matching line (with LINNUM or LINLABL) is found; carry is clear otherwise.

Error exits: None.

FINDLIN (routine number 22)

Description: FINDLIN is the same as FINDLIN0, except the search always starts with the beginning of the program.

On entry: LINNUM contains the line number to search for in the program or LINLABL contains the label to search for and LINNUM is zero.

On return: See FINDLIN0.

Error exits: None.

NOTNOW (routine number 24)

Description: Given an address in the pointer INDEX to a string descriptor, NOTNOW leaves the string length in the A register and an address to the actual string data in the pointer INDEX.

On entry: INDEX points to a string descriptor.

On return: INDEX points to the string data. A register contains the length of the string. Destroys A, X, and Y registers. Preserves the status register.

ERROR (routine number 26)

Description: ERROR raises the BASIC error condition given in the X register. If ON ERR is in effect and execution is in deferred mode, control is transferred to the BASIC program error handler. Otherwise, the error message is printed, and control is returned to direct mode. This routine resets the stack and never returns.

On entry: X register contains the BASIC error code.

On return: Never returns.

SERROR (routine number 28)

Description: SERROR translates the ProDOS errors shown below to the corresponding BASIC errors if they are in the table; otherwise, issues the ProDOS CALL ERROR = $ff.

On entry: A register is ProDOS return code.

On return: Never returns.

Table I-X

ProDOS error number	BASIC error number	BASIC error message
$10	81	DEVICE NOT FOUND
$11	82	INVALID DEVICE
$25	43	INT/FCB/VCB TBL FULL
$27	30	I/O
$28	42	DEVICE NOT CONNECTED
$2B	32	WRITE PROTECT
$2E	33	VOLUME SWITCHED
$2F	33	DRIVE EMPTY
$40	34	BAD PATH
$41	35	FILE TYPE
$42	43	INT/FCB/VCB TBL FULL
$44	36	PATH NOT FOUND
$45	37	VOLUME NOT FOUND
$46	35	FILE NOT FOUND
$47	38	DUPLICATE FILE
$48	39	DISK FULL
$49	44	DIRECTORY FULL
$4D	48	POSITION RANGE
$4E	40	FILE LOCKED
$4F	49	FILE CREATE
$50	26	FILE OPEN
$51	51	DAMAGED DIRECTORY
$52	27	VOLUME TYPE
$54	10	OUT OF MEMORY
$55	43	INT/FCB/VCB TBL FULL
$57	45	DUPLICATE VOLUME
$58	16	TYPE MISMATCH

FRECNOW (routine number 34)

Description: If you evaluate a string expression, the temporary string descriptor and string space must be freed after use if the string result is not going to be assigned to a string variable. FACPTR must point to the descriptor when this routine is called. FRECNOW should be called after using STRCP to allocate a temporary work string.

On entry: FACPTR points to string's descriptor.

On return: Descriptor deallocated if it was a temporary descriptor, and string pool space freed up if a temporary descriptor was deallocated.

Error exits: This routine does not take any error exits, but using it incorrectly could affect memory almost anywhere.

GETNAME (routine number 36)

Description: GETNAME is the filename setup routine used by IIGS BASIC for processing statement pathnames in either literal form (for immediate mode) or processing string expressions in deferred mode, and for creating a counted string, or P-string, in NAMBUF, a fixed address buffer used as input for FOPEN_AX and other routines.

On entry: TXTPTR points to the first byte of a string expression to be evaluated and converted into a ProDOS pathname in NAMBUF.

On return: The value of the string expression has been stored into NAMBUF with the count byte in byte 0. The address of NAMBUF is set up in the PTHPTR parameter used by FOPEN_AX to locate and open the file. TXTPTR points at the terminating character following the string expression. The terminating character should be either a colon, end-of-line ($00), or comma.

Error exits: The expression evaluator is called to evaluate a string expression, so any expression error, such a illegal quantity or overflow error, could occur.

FOPEN_AX (routine number 38)

Description: FOPEN_AX is the file open primitative in IIGS BASIC. It locates and opens a file at the ProDOS level, not at the IIGS BASIC file-number level. It is used by OPEN, LOAD, SAVE, CAT, INVOKE, RUN, CHAIN, and similar statements. When a file type is specified (X register <> 0), the file is created if it is not found. The memory-management routine UMSHRINK is called by FOPEN_AX if ProDOS can't allocate space for the ProDOS FCB and sector buffers.

❖ *Note:* FOPEN_AX can only be used to open a disk (block device) file; it cannot open character device files.

On entry: NAMBUF contains the pathname to locate, and the parameter list pointer, PTHPTR, contains the address of NAMBUF. PTHPTR is set by calling the GETNAME entry described above.

A register contains the open access request parameter as follows:

$01 = Read Only, $02 = Write Only, $03 = Read/Write

X register contains the open file type request parameter as follows:

$00 = Match any existing file with any file type

$01..$FF = Open only if its ProDOS file type = X-register

On return: A register returns the open access request, which is modified to be a $C3 if the file was created. X register will contain the ProDOS reference number for the file

Error exits: Most ProDOS 16 errors can occur, including:

- The access error ($4E) or file locked error
- The wrong type error ($41) or file type error
- The not found error ($46) or file not found error
- The file lost error ($4F) or file create error; this error occurs when a file is successfully created but then can't be opened immediately thereafter (a very unlikely sequence of events).

Refer to the *ProDOS 16 Reference* manual for all the errors that can occur for GETFILEINFO, OPEN, and CREATE to complete this list.

AY2XINT (routine number 40)

Description: AY2XINT converts the A and Y registers, input as a high-low signed 16-bit number, into a single integer in XACC with XTYPE, XCLAS, X.STS, and X.SGN, and so on, properly set.

On entry: A and Y registers are high-low signed 16-bit integer.

On return: XACC is set to single integer with the same value. XTYPE = 1 and XCLAS = $41.

Error exits: None.

POSINT (routine number 42)

Description: POSINT tests the status of XACC via the contents of X.SGN for a positive number, and then converts XACC to a single integer (XTYPE = 01), as if the BASIC construct CONV%(xacc) were called.

On entry: XACC, XTYPE, XCLAS, X.STS, and X.SGN set up for any numeric type.

On return: XACC, XACC+1 contains the 16-bit value if the number was in the range of 0 through 32767; XTYPE = 01 etc.

Error exits: The overflow error will result if the numeric value is outside the range for a 15-bit unsigned integer. The illegal quantity error will occur if X.SGN indicates that the number is negative.

CVTTXT2X (routine number 44)

Description: CVTTXT2X converts the ASCII string, pointed to by TXTPTR, into a binary numeric format in XACC. CVTTXT2X will always leave TXTPTR pointing to the first nonnumber character that it encounters. The conversion is done in two steps using the SANE Scanner function FCStr2Dec and then various FDEC2x decimal record to binary conversions. The SANE Scanner recognize +/- INF as the mathematical concept of infinity and the sequence NAN(digits) as an explicit request to generate a SANE Not a Number.

CVTTXT2X will, generate single or double integers, or single-, double- or extended-precision real results based on the number of significant digits and the presence or absence of a decimal fraction.

If the decimal form has more than nine digits or is not integral, a SANE extended real number is returned. If the decimal form is integral and nine or fewer digits, a double or single integer will be returned. Note that a long integer (more than nine digits) will be returned as extended and must be converted to the SANE COMP format to get a BASIC long integer.

If the decimal form has a fraction and seven or fewer significant digits, a single real is returned; if it has eight through sixteen significant digits, a SANE double real is returned. Otherwise, an extended real is returned.

There must be a nonfloating-point number terminator character to end the string. Number characters include the digits, the period, the plus and the minus signs, and the letters E and e.

On entry: TXTPTR points to ASCII representation of an integer or real number with up to 28 significant digits.

On return: XACC contains the binary integer or real format number. XTYPE, XCLAS, X.STS, X.SGN, and so on will be set accordingly. Carry will be set if a number was found and stored into XACC. Carry will be clear if a valid number was not found. TXTPTR will point to the first nonnumber character after the constant if a number was found, and remain unchanged otherwise.

Error exits: Some obscure SANE exceptions can occur, but generally almost all numeric conversion errors are handled by the preselection of the proper numeric type for the FDEC2x conversion.

DATAN (routine number 46)

Description: DATAN searches forward in the program for the end of the current statement. It stops when it finds the end-of-line token ($00) or a statement separator colon. DATAN does not use CHRGET. DATAN (which means DATA eNd) will properly skip over quoted strings (ignoring embedded colons) and all single and multibyte tokens.

On entry: TXTPTR points inside a statement at either an ASCII character or the first byte of a valid token (but not within a multibyte token).

On return: Y register contains the byte offset to the end of statement from the unchanged input TXTPTR.

Error exits: This routine will loop forever if a $00 or a $3A (:) does not exit within 256 bytes of the input TXTPTR.

STRCP (routine number 48)

Description: STRCP copies ASCII data into BASIC's string data pool and builds a temporary string descriptor pointing to the data.

This routine creates the data part of what BASIC calls a string, but it is a temporary one, not assigned to any variable. If this temporary string is not to be assigned to a variable (use STS2M for that) it must be freed up after being used, by using the FRECNOW routine.

Temporary string descriptors are allocated and built in a zero-page stack and must be deallocated in most recently allocated order. If you are allocating multiple temporary strings with STRCP, you must retain the pointers to their temporary descriptors and free them up in the proper order.

On entry: Y register contains the length of the string. Address of the text to be copied is in the pointer STRNG1.

On return: A temporary string descriptor is built (in zero page) and pointed to by FACPTR. XTYPE is set to $07, XCLAS is set to $87, and a pseudo-REFTBL value of $01 is stored into X.STS for use by the expression evaluator.

Error exits: The out of memory error exit could be taken.

STS2M (routine number 50)

Description: STS2M assigns a temporary string to a string variable or it duplicates an existing string and assigns the copy to the variable. This routine assumes that the address in FACPTR points to the descriptor of the result or source string to be assigned. FORPNT points to the destination variable's string descriptor. FORPNT is normally set by using PTRGET to locate the desired variable in the storage tables and then saving the address result in FORPNT.

STS2M sets the string pool back pointer via the address set up in HIGHDS by a call to STRCP or its own internal call executed when an existing variable's data was copied.

On entry: FORPNT points to the descriptor of the destination string variable. REFTBL is set up to indicate which table (simple, local or array) contains the destination variable. FACPTR points to the descriptor of the new value for the variable.

On return: Never returns.

Error exits: Variable error exit will be taken if the variable's descriptor is more than 64K from its storage table origin pointer.

STX2V (routine number 52)

Description: STX2V assigns a value to a variable. Takes the value in VTYPE, sets VCLAS accordingly, and then uses the result in XACC and flags XTYPE, XCLAS to select either numeric or string assignment. The variable is defined by the pointer FORPNT and the REFTBL flag, and STX2V assigns XACC to the variable. The various XACC data formats and their XTYPEs are listed earlier in the description of PTRGET (routine number 06). This routine does not validate its input data and could store values anywhere in memory. Use it carefully!

On entry: FORPNT points to the variable; VTYPE and REFTBL describe the variable's type (VTYPE = XTYPE) and base address. XACC has the variable's new value, or FACPTR points to a string descriptor. XTYPE, XCLAS, X.STS, and so on define the type of the value in XACC. If the variable is numeric and XTYPE and VTYPE are different, numeric type conversion is performed to the type of the variable (given by VTYPE) before assigning the result.

On return: Never returns.

Error exits: If XTYPE and VTYPE flags do not reflect the actual type of the variable at VARPNT, the value will be stored in whatever is there as though it was the proper variable type. In addition, if XACC does not contain the type of data specified by XTYPE, whatever is in XACC will be assigned as though it were correct.

Numeric type conversion may generate overflow or other SANE errors, and string assignment may cause garbage collection or an out of memory error.

CONV2STR (routine number 54)

Description: CONV2STR takes XACC and converts it to an ASCII string for any valid XTYPE. SIt is the same as the CONV$ function in BASIC. If XACC is a real, then a STR$-like operation is done, leaving a pointer to the string descriptor in FACPTR. This routine does nothing if XTYPE already is a string XTYPE ($07). XACC can contain any valid numeric type from XTYPE = $00 through $06. If CONV2STR is called with an integer (single, double, or long), the number is converted to a string with all the digits of the integer's value.

On entry: XACC has some value determined by XTYPE, XCLASS, and so on.

On return: XACC now has that value expressed in string form XTYPE = 07, XCLAS = $87 and FACPTR will point to a temporary string descriptor.

Error exits: None. The data in XACC must correctly match the given XTYPE.

Appendix J

Standard file types

Table J-1

Mnemonic code	File type	Utility descriptor	Description
UNK	$00	Unknown	Uncategorized file (SOS)
BAD	$01	BadBlocks	Bad block file
PCD	$02	PascalCode	Pascal code file
PTX	$03	PascalText	Pascal text file
TXT	$04	AsciiText	ASCII text file (SOS and ProDOS)
PDA	$05	PascalData	Pascal data file
BIN	$06	BinaryFile	General binary file (SOS and ProDOS)
FNT	$07	/// Fontfile	SOS font file
FOT	$08	Graphicfile	SOS Foto file
BA3	$09	/// BasicProg	Business BASIC program file
DA3	$0A	///BasicData	Business BASIC data file
WPF	$0B	///WordProc	Word processor file
SOS	$0C	///SOSFile	SOS system file
$0D	$0D		Reserved
$0E	$0E		Reserved
DIR	$0F	Directory	Subdirectory file (SOS and ProDOS)
RPD	$10	/// RPSData	RPS data file
RPI	$11	///RPSIndex	RPS index file
AFD	$12	///AppleFile	AppleFile discard file
AFM	$13	///AppleFile	AppleFile model file
AFR	$14	///AppleFile	AppleFile report format file
SCL	$15		Screen library file
ADB	$19	AppleWorks DB	AppleWorks data base file
AWP	$1A	AppleWorks WP	AppleWorks word processor file
ASP	$1B	AppleWorks SS	AppleWorks spreadsheet file
GSB	$AB		IIGS BASIC program file
TDF	$AC		IIGS BASIC toolbox definition file
BDF	$AD		IIGS BASIC data file

Table J-1

Mnemonic code	File type	Utility descriptor	Description
SRC	$B0	APWTextfile	APW text file
OBJ	$B1	APWCodefile	APW object file
LIB	$B2	APWLibrary	APW library file
S16	$B3	System16	ProDOS 16 application file (OMF load)
RTL	$B4		APW run-time library file
EXE	$B5		APW shell application file
PPI	$B6		ProDOS 16 permanent Init file
PTI	$B7		ProDOS 16 temporary Init file
NDA	$B8		New desk accessory
CDA	$B9	DeskAccs	Classic desk accessory
TOL	$BA	SystemTool	ProDOS 16 tool set file
DVR	$BB		ProDOS 16 driver file
$BC	$BC		Reserved for OMF
$BD	$BD		Reserved for OMF
$BE	$BE		Reserved for OMF
DOC	$BF		ProDOS 16 document file
$C0	$C0		?? what is this one??
PIC	$C1		Picture file (Super Hi-Res FOT?)
$E0	$E0		
WAV	$E1		IIGS BASIC Wavebank data file
DTS	$E2		ProDOS 16 ??? file
R16	$EE		EDASM 816 relative object file
PAS	$EF		Pascal area on a partitioned volume
CMD	$F0	ProDOS-CI	ProDOS 8 CI added command file
DSK	$F1		
O.S	$F9	ProDOS-16	ProDOS 16 operating system
INT	$FA	IntegerProg	Integer BASIC program file
IVR	$FB	IntegerVar	Integer BASIC variable file
BAS	$FC	AplSoftProg	AppleSoft program file
VAR	$FD	AplSoftVar	AppleSoft variable file
REL	$FE	RelocatableCode	EDASM relocatable code file
SYS$	FF	ProDOS-System	ProDOS 8 system program file

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using the Apple Macintosh™ Plus and Microsoft® Word. Proof and final pages were created on the Apple LaserWriter® Plus. POSTSCRIPT™, the LaserWriter page-description language, was developed by Adobe Systems Incorporated.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Program listings are set in Apple Courier, a monospaced font.

GSBASIC Release Notes

Version 1.0 Beta 4

September 10, 1987

Remember, this is a BETA software release – useful but not 100% reliable or bug free. Take appropriate precautions by saving your source programs frequently.

Important Information

GSBASIC requires a minimum of 96K free memory (including 1 whole free 64K bank for the interpreter, currently 59000+ bytes, plus a 32K data segment, plus 14 bank zero pages) to boot. The standard GSB.HELLO program attempts to expand the data segment to 65K bytes, thus increasing the minimun default requirement to 131+K bytes. The memory management and allocations in the minimum 512K system will just support a 100K data segment and allow a reasonable size program.

The Desktop environment toolbox requires an additional 192K for the toolsets, the LIBRARY dictionaries and toolset memory requirements. The Desktop demo program, PICS, will not run in a 512K system.

Known Bugs

Cosmetic features, such as indentation of FOR .. NEXT loops and spacing after keywords, are not finalized, although program execution is unaffected.

The display of a pathname is truncated to 40 characters when displayed by the CAT command.

Scrolling back in the command buffer to display lines previously entered will sometimes result in garbage displays if long (>80 column) lines have been entered.

Known Limitations

The interpreter returns an integer value of -32768 when an out of range real or larger integer is converted to a single integer IF you are using the original ROMS. SANE in the initial release ROMS does not properly report the INVALID exception back to the interpreter and so the expected

?OVERFLOW ERROR does not occur. SANE has been fixed in the GS ROM update.

GSBASIC requires ProDOS 16 V1.2 or greater (v1.3 is on the GSBASIC distribution disk). The new GetDirEntry function is required by the CATALOG,CAT, and DIR commands. The message ?ProDOS VERSION error will occur if the interpreter is run under an earlier version of ProDOS 16 that doesn't have the GetDirEntry function call.

To make BASIC programs launchable from the finder, the GSBASIC.SYS16 file must reside in the top-level directory of the boot volume.

Documentation Corrections

DATA is not allowed as an alias for the BDF **filtyp** in I/O commands.

Chapter 8 of the manual incorrectly refers to UIR() as UIR%().

GSBASIC recognizes a startup filetype for EXEC as well as RUN. Thus GSB.HELLO can be a SRC file with an AUXID of $BA and GSBASIC will execute at startup as if 'EXEC GSB.HELLO' were entered instead of 'RUN GSB.HELLO'.

The DIR command displays in NORMAL characters. The User is responsible for restoring INVERSE (if needed) after the DIR command has been executed.

Tool Programming

Tool programming from GSBASIC is not for everybody. The GSBASIC reference manual does not provide any tutorial information on tool programming; the Apple IIGS Tool Reference Manual is absolutely indispensible. In general tool call parameters are coded as you would expect given the Tool Reference Manual description. However, in some cases it may be unclear how certain data types were implemented in the GSBASIC TDF's. These release notes include a listing of the text files that were used to create the TDF's. Their format is described in the TDFBUILD appendix. The TDFBUILD utility itself is not available with this release.

The DEMOS subdirectory contains five files. Four of them are files with picture data. They are used by the GSBASIC program PICS, which is also this subdirectory.

The program PICS is a program that has been used as a demo in other GS programming environments. It is a simple program that opens up four windows and loads pictures into them. It supports all window functions, including scrolling, dragging (reposition the window), resizing, zooming clipping, hiding, and reshowing windows.

Here is a description of how the program works, from the top to the botto

- points prefix 6 at the tdf's
- clears the screen.
- grabs a large chunk of memory.
- set up the data structures that will be used later.
- set up titles for window with Pascal-type strings, acces via the =^
- loads the quickdraw tdf-file
- calls the procedure "please.wait", which displays just t on the screen. Startup takes so long because big data structures are being initialized interpretively.
- load up the data structures by restoring the data pointe a certain line, and calling the procedure "readtable", whic loads the data from the data statements, and puts it into the byte-arrays for use in tool calls.
- set the window titles
- load the tdf files if the library file was not present
- set up error trapping
- starts up the memory manager and gets a process id.
- start up managers
- set up the menus
- set up the four graphics windows
- draw a window and put text "Loading Pictures" in it
- call the procedure to load the pictures
- hide that window
- make window 1 the current window
- turn off the quit flag
- set up the menu handling routines: about, doquit, mw mw2, mw3, and mw4
- set up the event handling routine for hiding a window
- turn on TASKPOLL with an event mask of $FFF

the "GetNextEvent" loop; loops until the user chooses quit
- the About procedure. Turns on the text screen and displays a cute message
- the Quit procedure. Sets the quit flag to 1 and turns off TASKPOLL so that no other events are trapped.

the Window procedures: brings the appropriate window to the front and unhilites the menu bar when done.
- Hides the selected window
- the Please.Wait procedure. Starts up graphics and prints out "One Moment Please..."
- the Msg procedure. Clears the screen and prints the string passed as a parameter
- the Deref function. Gets the word value from the address passed as a parameter
- the DoErr handler. Turns off error trapping and falls into the ShutDown handler.
- the ShutDown handler. Shuts down all of the managers, sets the screen to text, and ends the program.
- the PWin procedure. Copies a picture from an array into a window's Grafport.
- the ReadTable function. Calls DoSet to read data statements and load the data into byte arrays. Returns a pointer to the array.
- the DoSet function. Reads data statements and translates the data found into appropriately sized bytes to put into byte arrays. Has routines to handle integers, double-integers, double-precision reals, and strings. You might want to use this routine to initialize data structures in your programs. It is unbearably slow and desperately wants to be written as an invokable module. It is not the only data-initialization scheme available; it's virtue is that it allows you to easily change your data structure. Once you've got them defined, you may want to create a data file and then simply load it in from your program.
- the data for the menus
- the data for a dialog
- the data for the four windows
- miscellaneous data (size of window,...)
- the draw routines for the four windows
- the LoadPics procedure. Loads the four pictures from disk and stores them into byte arrays.

Note: it is the programmer's responsibility to correctly shut down all tools that have been initialized by an application. Failure to correctly shut down tools can result in unpredictable behavior after BASIC has been exited.

BUG REPORT

Written bug reports, describing completely REPEATABLE bugs, with example source listings, should be sent to: GSBASIC Product Management, MS 27-S, Apple Computer, Inc., 20525 Mariani Avenue, Cupertino, CA 95014 If source code required to demonstrate bug is lengthy, inclusion of disk copy is appreciated.

Problem:

Name, Address, Telephone Number:

Comments and suggestions are also welcome.

```
H,L,$0100,3,MiscTool
P,3,1,0,B,N,00,MTBootInit
P,3,2,0,B,N,00,MTStartUp
P,3,3,0,B,N,00,MTShutDown
F,3,4,1,B,I,00,MTVersion
P,3,5,0,B,N,00,MTReset
F,3,6,1,B,I,00,MTStatus
P,3,28,0,B,N,04,I,I,I,I,ClampMouse
P,3,27,0,B,N,00,ClearMouse
P,3,20,0,B,N,00,ClrHeartBeat
P,3,33,0,B,N,01,I,DeleteID
P,3,19,0,B,N,01,A,DelHeartBeat
P,3,36,4,B,N,04,I,I,I,I,FWEntry
P,3,43,4,B,N,00,GetAbsClamp
F,3,22,2,B,A,01,I,GetAddr
F,3,41,1,B,I,00,GetIRQEnable
P,3,29,4,B,N,00,GetMouseClamp
F,3,32,1,B,I,01,I,GetNewID
F,3,37,2,B,L,00,GetTick
F,3,17,2,B,A,01,I,GetVector
P,3,26,0,B,N,00,HomeMouse
P,3,24,0,B,N,01,I,InitMouse
P,3,35,0,B,N,01,I,IntSource
F,3,40,1,B,I,07,A,A,A,I,A,I,A,Munger
F,3,38,1,B,I,04,A,A,A,I,PackBytes
P,3,30,0,B,N,02,I,I,PosMouse
P,3,15,0,B,N,01,A,ReadAsciiTime
F,3,12,1,B,I,01,I,ReadBParam
P,3,10,0,B,N,01,A,ReadBRam
P,3,23,3,B,N,00,ReadMouse
P,3,13,4,B,N,01,A,ReadTimeHex
F,3,31,1,B,I,00,ServeMouse
P,3,42,0,B,N,04,I,I,I,I,SetAbsClamp
P,3,18,0,B,N,01,A,SetHeartBeat
P,3,25,0,B,N,01,I,SetMouse
P,3,16,0,B,N,02,I,A,SetVector
P,3,34,0,B,N,01,I,StatusID
P,3,44,0,B,N,00,SysBeep
P,3,21,0,B,N,02,I,A,SysFailMgr
F,3,39,1,B,I,04,A,I,A,A,UnPackBytes
P,3,11,0,B,N,02,I,I,WriteBParam
P,3,9,0,B,N,01,A,WriteBRam
P,3,14,0,B,N,3,I,I,I,WriteTimeHex
```

```
H,L,$0100,4,Quickdraw
P,4,1,0,B,N,00,QDBootInit
P,4,2,0,B,N,04,I,I,I,I,QDStartUp
P,4,3,0,B,N,00,QDShutDown
F,4,4,1,B,I,00,QDVersion
P,4,5,0,B,N,00,QDReset
F,4,6,1,B,I,00,QDStatus
P,4,128,0,B,N,02,A,A,AddPt
P,4,172,0,B,N,02,I,A,CharBounds
F,4,168,1,B,I,01,I,CharWidth
P,4,21,0,B,N,01,I,ClearScreen
P,4,38,0,B,N,01,A,ClipRect
P,4,185,0,B,N,00,ClosePicture
P,4,194,0,B,N,00,ClosePoly
P,4,26,0,B,N,01,A,ClosePort
P,4,110,0,B,N,01,A,CloseRgn
P,4,105,0,B,N,02,A,A,CopyRgn
P,4,174,0,B,N,02,A,A,CStringBounds
F,4,170,1,B,I,01,A,CStringWidth
P,4,115,0,B,N,03,A,A,A,DiffRgn
P,4,104,0,B,N,01,A,DisposeRgn
P,4,164,0,B,N,01,I,DrawChar
P,4,166,0,B,N,01,A,DrawCString
P,4,186,0,B,N,02,A,A,DrawPicture
P,4,165,0,B,N,01,A,DrawString
P,4,167,0,B,N,02,A,I,DrawText
F,4,82,1,B,I,01,A,EmptyRect
F,4,120,1,B,I,01,A,EmptyRgn
F,4,131,1,B,I,02,A,A,EqualPt
F,4,81,1,B,I,02,A,A,EqualRect
F,4,119,1,B,I,02,A,A,EqualRgn
P,4,100,0,B,N,03,A,I,I,EraseArc
P,4,90,0,B,N,01,A,EraseOval
P,4,190,0,B,N,01,A,ErasePoly
P,4,85,0,B,N,01,A,EraseRect
P,4,123,0,B,N,01,A,EraseRgn
P,4,95,0,B,N,03,A,I,I,EraseRRect
P,4,102,0,B,N,04,A,I,I,A,FillArc
P,4,92,0,B,N,02,A,A,FillOval
P,4,192,0,B,N,02,A,A,FillPoly
P,4,87,0,B,N,02,A,A,FillRect
P,4,125,0,B,N,02,A,A,FillRgn
P,4,97,0,B,N,04,A,I,I,A,FillRRect
P,4,204,0,B,N,03,I,I,I,ForceBufDims
P,4,98,0,B,N,03,A,I,I,FrameArc
P,4,88,0,B,N,01,A,FrameOval
P,4,188,0,B,N,01,A,FramePoly
P,4,83,0,B,N,01,A,FrameRect
P,4,121,0,B,N,01,A,FrameRgn
P,4,93,0,B,N,03,A,I,I,FrameRRect
F,4,9,2,B,A,01,I,GetAddress
F,4,177,1,B,I,00,GetArcRot
F,4,163,1,B,I,00,GetBackColor
P,4,53,0,B,N,01,A,GetBackPat
F,4,213,2,B,L,00,GetCharExtra
P,4,37,0,B,N,01,A,GetClip
F,4,199,2,B,A,00,GetClipHandle
F,4,17,1,B,I,02,I,I,GetColorEntry
P,4,15,0,B,N,02,I,A,GetColorTable
F,4,143,2,B,A,00,GetCursorAdr
F,4,207,1,B,I,00,GetFGSize
F,4,149,2,B,A,00,GetFont
F,4,153,1,B,I,00,GetFontFlags
P,4,151,0,B,N,01,A,GetFontGlobals
```

```
P,4,209,2,B,N,00,GetFontID
P,4,150,0,B,N,01,A,GetFontInfo
F,4,217,1,B,I,02,A,I,GetFontLore
F,4,161,1,B,I,00,GetForeColor
F,4,69,2,B,A,00,GetGrafProcs
F,4,23,1,B,I,00,GetMasterSCB
P,4,41,0,B,N,01,A,GetPen
P,4,51,0,B,N,01,A,GetPenMask
F,4,47,1,B,I,00,GetPenMode
P,4,49,0,B,N,01,A,GetPenPat
P,4,45,0,B,N,01,A,GetPenSize
P,4,43,0,B,N,01,A,GetPenState
F,4,63,2,B,L,00,GetPicSave
F,4,136,1,B,I,02,I,I,GetPixel
F,4,67,2,B,L,00,GetPolySave
F,4,28,2,B,A,00,GetPort
P,4,30,0,B,N,01,A,GetPortLoc
P,4,32,0,B,N,01,A,GetPortRect
F,4,65,2,B,L,00,GetRgnSave
P,4,216,0,B,N,01,A,GetROMFont
F,4,19,1,B,I,01,I,GetSCB
F,4,159,2,B,L,00,GetSpaceExtra
F,4,12,1,B,I,00,GetStandardSCB
F,4,73,2,B,L,00,GetSysField
F,4,179,2,B,A,00,GetSysFont
F,4,155,1,B,I,00,GetTextFace
F,4,157,1,B,I,00,GetTextMode
F,4,211,1,B,I,00,GetTextSize
F,4,71,2,B,L,00,GetUserField
F,4,201,2,B,A,00,GetVisHandle
P,4,181,0,B,N,01,A,GetVisRgn
P,4,133,0,B,N,01,A,GlobalToLocal
P,4,11,0,B,N,00,GrafOff
P,4,10,0,B,N,00,GrafOn
P,4,144,0,B,N,00,HideCursor
P,4,39,0,B,N,00,HidePen
P,4,215,0,B,N,02,I,I,InflateTextBuffer
P,4,13,0,B,N,01,A,InitColorTable
P,4,202,0,B,N,00,InitCursor
P,4,25,0,B,N,01,A,InitPort
P,4,76,0,B,N,03,A,I,I,InsetRect
P,4,112,0,B,N,03,A,I,I,InsetRgn
P,4,101,0,B,N,03,A,I,I,InvertArc
P,4,91,0,B,N,01,A,InvertOval
P,4,191,0,B,N,01,A,InvertPoly
P,4,86,0,B,N,01,A,InvertRect
P,4,124,0,B,N,01,A,InvertRgn
P,4,96,0,B,N,03,A,I,I,InvertRRect
P,4,187,0,B,N,01,A,KillPicture
P,4,195,0,B,N,01,A,KillPoly
P,4,61,0,B,N,02,I,I,Line
P,4,60,0,B,N,02,I,I,LineTo
P,4,132,0,B,N,01,A,LocalToGlobal
P,4,197,0,B,N,03,A,A,A,MapPoly
P,4,138,0,B,N,03,A,A,A,MapPt
P,4,139,0,B,N,03,A,A,A,MapRect
P,4,140,0,B,N,03,A,A,A,MapRgn
P,4,59,0,B,N,02,I,I,Move
P,4,34,0,B,N,02,I,I,MovePortTo
P,4,58,0,B,N,02,I,I,MoveTo
F,4,103,2,B,A,00,NewRgn
F,4,82,1,B,I,01,A,NotEmptyRect
P,4,146,0,B,N,00,ObscureCursor
P,4,196,0,B,N,03,A,I,I,OffsetPoly
```

```
P,4,75,0,B,N,03,A,I,I,OffsetRect
P,4,111,0,B,N,03,A,I,I,OffsetRgn
F,4,183,2,B,A,01,A,OpenPicture
F,4,193,2,B,A,00,OpenPoly
P,4,24,0,B,N,01,A,OpenPort
P,4,109,0,B,N,00,OpenRgn
P,4,99,0,B,N,03,A,I,I,PaintArc
P,4,89,0,B,N,01,A,PaintOval
P,4,127,0,B,N,01,A,PaintPixels
P,4,189,0,B,N,01,A,PaintPoly
P,4,84,0,B,N,01,A,PaintRect
P,4,122,0,B,N,01,A,PaintRgn
P,4,94,0,B,N,03,A,I,I,PaintRRect
P,4,54,0,B,N,00,PenNormal
P,4,184,0,B,N,03,I,I,A,PicComment
P,4,214,0,B,N,05,A,A,I,I,I,PPToPort
P,4,80,0,B,N,03,A,A,A,Pt2Rect
F,4,79,1,B,I,02,A,A,PtInRect
F,4,117,1,B,I,02,A,A,PtInRgn
F,4,134,1,B,I,00,Random
F,4,118,1,B,I,02,A,A,RectInRgn
P,4,108,0,B,N,02,A,A,RectRgn
P,4,206,0,B,N,01,A,RestoreBufDims
P,4,205,0,B,N,01,A,SaveBufDims
P,4,137,0,B,N,03,A,A,A,ScalePt
P,4,126,0,B,N,04,A,I,I,A,ScrollRect
F,4,77,1,B,I,03,A,A,A,SectRect
P,4,113,0,B,N,03,A,A,A,SectRgn
P,4,20,0,B,N,01,I,SetAllSCBs
P,4,176,0,B,N,00,SetArcRot
P,4,162,0,B,N,01,I,SetBackColor
P,4,52,0,B,N,01,A,SetBackPat
P,4,203,0,B,N,03,I,I,I,SetBufDims
P,4,212,0,B,N,01,L,SetCharExtra
P,4,36,0,B,N,01,A,SetClip
P,4,198,0,B,N,01,A,SetClipHandle
P,4,16,0,B,N,03,I,I,I,SetColorEntry
P,4,14,0,B,N,02,I,A,SetColorTable
P,4,142,0,B,N,01,A,SetCursor
P,4,106,0,B,N,01,A,SetEmptyRgn
P,4,148,0,B,N,01,A,SetFont
P,4,152,0,B,N,01,I,SetFontFlags
P,4,208,0,B,N,01,L,SetFontID
P,4,160,0,B,N,01,I,SetForeColor
P,4,68,0,B,N,01,A,SetGrafProcs
P,4,182,0,B,N,01,I,SetIntUse
P,4,22,0,B,N,01,I,SetMasterSCB
P,4,35,0,B,N,02,I,I,SetOrigin
P,4,50,0,B,N,01,A,SetPenMask
P,4,46,0,B,N,01,I,SetPenMode
P,4,48,0,B,N,01,A,SetPenPat
P,4,44,0,B,N,02,I,I,SetPenSize
P,4,42,0,B,N,01,A,SetPenState
P,4,62,0,B,N,01,L,SetPicSave
P,4,66,0,B,N,01,L,SetPolySave
P,4,27,0,B,N,01,A,SetPort
P,4,29,0,B,N,01,A,SetPortLoc
P,4,31,0,B,N,01,A,SetPortRect
P,4,33,0,B,N,02,I,I,SetPortSize
P,4,130,0,B,N,03,A,I,I,SetPt
P,4,135,0,B,N,01,L,SetRandSeed
P,4,74,0,B,N,05,A,I,I,I,I,SetRect
P,4,107,0,B,N,05,A,I,I,I,I,SetRectRgn
P,4,64,0,B,N,01,L,SetRgnSave
```

```
P,4,18,0,B,N,02,I,I,SetSCB
P,4,56,0,B,N,01,I,SetSolidBackPat
P,4,55,0,B,N,01,I,SetSolidPenPat
P,4,158,0,B,N,01,L,SetSpaceExtra
P,4,141,0,B,N,01,A,SetStdProcs
P,4,72,0,B,N,01,L,SetSysField
P,4,178,0,B,N,01,A,SetSysFont
P,4,154,0,B,N,01,I,SetTextFace
P,4,156,0,B,N,01,I,SetTextMode
P,4,210,0,B,N,01,I,SetTextSize
P,4,70,0,B,N,01,L,SetUserField
P,4,200,0,B,N,01,A,SetVisHandle
P,4,180,0,B,N,01,A,SetVisRgn
P,4,145,0,B,N,00,ShowCursor
P,4,40,0,B,N,00,ShowPen
P,4,57,0,B,N,02,A,I,SolidPattern
P,4,173,0,B,N,02,A,A,StringBounds
F,4,169,1,B,I,01,A,StringWidth
P,4,129,0,B,N,02,A,A,SubPt
P,4,175,0,B,N,03,A,I,A,TextBounds
F,4,171,1,B,I,02,A,I,TextWidth
P,4,78,0,B,N,03,A,A,A,UnionRect
P,4,114,0,B,N,03,A,A,A,UnionRgn
P,4,116,0,B,N,03,A,A,A,XorRgn
```

```
H,L,$0100,7,Scheduler
P,7,1,0,B,N,00,SchBootInit
P,7,2,0,B,N,00,SchStartUp
P,7,3,0,B,N,00,SchShutDown
F,7,4,1,B,I,00,SchVersion
P,7,5,0,B,N,00,SchReset
F,7,6,1,B,I,00,SchStatus
F,7,9,1,B,I,01,A,SchAddTask
P,7,10,0,B,N,00,SchFlush
```

```
H,L,$0100,8,Sound
P,8,1,0,B,N,00,SoundBootInit
P,8,2,0,B,N,01,I,SoundStartUp
P,8,3,0,B,N,00,SoundShutDown
F,8,4,1,B,I,00,SoundVersion
P,8,5,0,B,N,00,SoundReset
F,8,6,1,B,I,00,SoundToolStatus
F,8,17,1,B,I,01,I,FFGeneratorStatus
F,8,20,1,B,I,01,I,FFSoundDoneStatus
F,8,16,1,B,I,00,FFSoundStatus
P,8,14,0,B,N,02,I,A,FFStartSound
P,8,15,0,B,N,01,I,FFStopSound
F,8,12,1,B,I,01,I,GetSoundVolume
F,8,11,2,B,A,00,GetTableAddress
P,8,10,0,B,N,03,A,I,I,ReadRamBlock
P,8,18,0,B,N,01,L,SetSoundMIRQV
P,8,13,0,B,N,02,I,I,SetSoundVolume
F,8,19,2,B,A,01,L,SetUserSoundIRQV
P,8,9,0,B,N,03,A,I,I,WriteRamBlock
```

```
H,L,$0100,10,SANE
P,10,1,0,B,N,00,SANEBootInit
P,10,2,0,B,N,00,SANEStartUp
P,10,3,0,B,N,00,SANEShutDown
F,10,4,1,B,I,00,SANEVersion
P,10,5,0,B,N,00,SANEReset
F,10,6,1,B,I,00,SANEStatus
P,10,10,0,B,N,00,SANEDecStr816
P,10,11,0,B,N,00,SANEElems816
P,10,9,0,B,N,00,SANEFP816
```

```
H,L,$0100,12,TextTool
P,12,1,0,B,N,00,TextBootInit
P,12,2,0,B,N,00,TextStartUp
P,12,3,0,B,N,00,TextShutDown
F,12,4,1,B,I,00,TextVersion
P,12,5,0,B,N,00,TextReset
F,12,6,1,B,I,00,TextStatus
P,12,22,0,B,N,02,I,I,CtlTextDev
P,12,31,0,B,N,03,A,I,I,ErrWriteBlock
P,12,25,0,B,N,01,I,ErrWriteChar
P,12,33,0,B,N,01,A,ErrWriteCString
P,12,27,0,B,N,01,A,ErrWriteLine
P,12,29,0,B,N,01,A,ErrWriteString
P,12,14,2,B,N,00,GetErrGlobals
P,12,20,3,B,N,00,GetErrorDevice
P,12,12,2,B,N,00,GetInGlobals
P,12,18,3,B,N,00,GetInputDevice
P,12,13,2,B,N,00,GetOutGlobals
P,12,19,3,B,N,00,GetOutputDevice
P,12,21,0,B,N,01,I,InitTextDev
F,12,34,1,B,I,01,I,ReadChar
F,12,36,1,B,I,04,A,I,I,I,ReadLine
P,12,11,0,B,N,02,I,I,SetErrGlobals
P,12,17,0,B,N,02,I,L,SetErrorDevice
P,12,9,0,B,N,02,I,I,SetInGlobals
P,12,15,0,B,N,02,I,L,SetInputDevice
P,12,10,0,B,N,02,I,I,SetOutGlobals
P,12,16,0,B,N,02,I,L,SetOutputDevice
P,12,23,0,B,N,02,I,I,StatusTextDev
F,12,35,1,B,I,04,A,I,I,I,TextReadBlock
P,12,30,0,B,N,03,A,I,I,TextWriteBlock
P,12,24,0,B,N,01,I,WriteChar
P,12,32,0,B,N,01,A,WriteCString
P,12,26,0,B,N,01,A,WriteLine
P,12,28,0,B,N,01,A,WriteString
```

```
H,L,$0100,14,Window
P,14,1,0,B,N,00,WindBootInit
P,14,2,0,B,N,01,I,WindStartUp
P,14,3,0,B,N,00,WindShutDown
F,14,4,1,B,I,00,WindVersion
P,14,5,0,B,N,00,WindReset
F,14,6,1,B,I,00,WindStatus
P,14,30,0,B,N,01,A,BeginUpdate
P,14,36,0,B,N,01,A,BringToFront
F,14,10,1,B,I,01,A,CheckUpdate
P,14,11,0,B,N,01,A,CloseWindow
F,14,12,2,B,A,02,I,L,Desktop
P,14,26,0,B,N,06,I,I,I,I,A,A,DragWindow
P,14,81,0,B,N,00,EndInfoDrawing
P,14,31,0,B,N,01,A,EndUpdate
F,14,23,1,B,I,03,A,I,I,FindWindow
F,14,21,2,B,A,00,FrontWindow
F,14,72,2,B,A,01,A,GetContentDraw
P,14,62,1,B,N,01,A,GetContentOrigin
F,14,47,2,B,A,01,A,GetContentRgn
F,14,64,2,B,L,01,A,GetDataSize
F,14,49,2,B,A,01,A,GetDefProc
F,14,82,2,B,A,00,GetFirstWindow
P,14,16,0,B,N,02,A,A,GetFrameColor
F,14,74,2,B,A,01,A,GetInfoDraw
F,14,53,2,B,L,01,A,GetInfoRefCon
F,14,66,2,B,L,01,A,GetMaxGrow
F,14,42,2,B,A,01,A,GetNextWindow
F,14,70,2,B,L,01,A,GetPage
P,14,79,0,B,N,02,A,A,GetRectInfo
F,14,68,2,B,L,01,A,GetScroll
F,14,46,2,B,A,01,A,GetStructRgn
F,14,76,1,B,I,01,A,GetSysWFlag
F,14,48,2,B,A,01,A,GetUpdateRgn
F,14,51,2,B,A,01,A,GetWControls
F,14,44,1,B,I,01,A,GetWFrame
F,14,43,1,B,I,01,A,GetWKind
F,14,32,2,B,A,00,GetWMgrPort
F,14,41,2,B,L,01,A,GetWRefCon
F,14,14,2,B,A,01,A,GetWTitle
F,14,55,2,B,A,01,A,GetZoomRect
F,14,27,2,B,L,05,I,I,I,I,A,GrowWindow
P,14,18,0,B,N,01,A,HideWindow
P,14,34,0,B,N,02,I,A,HiliteWindow
P,14,58,0,B,N,01,A,InvalRect
P,14,59,0,B,N,01,A,InvalRgn
P,14,25,0,B,N,03,I,I,A,MoveWindow
F,14,9,2,B,A,01,A,NewWindow
P,14,33,1,B,N,03,I,I,A,PinRect
P,14,57,0,B,N,01,A,RefreshDesktop
P,14,17,0,B,N,01,A,SelectWindow
P,14,20,0,B,N,02,A,A,SendBehind
P,14,73,0,B,N,02,A,A,SetContentDraw
P,14,63,0,B,N,03,I,I,A,SetContentOrigin
P,14,65,0,B,N,03,I,I,A,SetDataSize
P,14,50,0,B,N,02,A,A,SetDefProc
P,14,15,0,B,N,02,A,A,SetFrameColor
P,14,22,0,B,N,02,A,A,SetInfoDraw
P,14,54,0,B,N,02,L,A,SetInfoRefCon
P,14,67,0,B,N,03,I,I,A,SetMaxGrow
P,14,52,0,B,N,02,I,A,SetOriginMask
P,14,71,0,B,N,03,I,I,A,SetPage
P,14,69,0,B,N,03,I,I,A,SetScroll
P,14,75,0,B,N,01,A,SetSysWindow
```

```
P,14,45,0,B,N,02,I,A,SetWFrame
F,14,78,2,B,A,01,A,SetWindowIcons
P,14,40,0,B,N,02,L,A,SetWRefCon
P,14,13,0,B,N,02,A,A,SetWTitle
P,14,56,0,B,N,02,A,A,SetZoomRect
P,14,35,0,B,N,02,I,A,ShowHide
P,14,19,0,B,N,01,A,ShowWindow
P,14,28,0,B,N,03,I,I,A,SizeWindow
P,14,77,0,B,N,01,A,StartDrawing
P,14,80,0,B,N,02,A,A,StartInfoDrawing
F,14,29,1,B,I,02,I,A,TaskMaster
F,14,24,1,B,I,03,I,I,A,TrackGoAway
F,14,38,1,B,I,03,I,I,A,TrackZoom
P,14,60,0,B,N,01,A,ValidRect
P,14,61,0,B,N,01,A,ValidRgn
F,14,83,2,B,L,09,A,A,I,I,A,A,A,I,I,WindDragRect
P,14,37,0,B,N,00,WindNewRes
F,14,86,1,B,I,00,WindowGlobal
P,14,39,0,B,N,01,A,ZoomWindow
```

```
H,L,$0100,15,Menu
P,15,1,0,B,N,00,MenuBootInit
P,15,2,0,B,N,02,I,I,MenuStartUp
P,15,3,0,B,N,00,MenuShutDown
F,15,4,1,B,I,00,MenuVersion
P,15,5,0,B,N,00,MenuReset
F,15,6,1,B,I,00,MenuStatus
P,15,28,0,B,N,03,I,I,I,CalcMenuSize
P,15,50,0,B,N,02,I,I,CheckMItem
F,15,20,1,B,I,01,I,CountMItems
P,15,14,0,B,N,01,I,DeleteMenu
P,15,16,0,B,N,01,I,DeleteMItem
P,15,49,0,B,N,01,I,DisableMItem
P,15,46,0,B,N,01,A,DisposeMenu
P,15,42,0,B,N,00,DrawMenuBar
P,15,48,0,B,N,01,I,EnableMItem
F,15,19,1,B,I,00,FixMenuBar
P,15,12,0,B,N,00,FlashMenuBar
F,15,24,2,B,L,00,GetBarColors
F,15,10,2,B,A,00,GetMenuBar
F,15,32,1,B,I,01,I,GetMenuFlag
F,15,27,2,B,A,00,GetMenuMgrPort
F,15,34,2,B,A,01,I,GetMenuTitle
F,15,22,2,B,A,01,I,GetMHandle
F,15,37,2,B,A,01,I,GetMItem
F,15,39,1,B,I,01,I,GetMItemFlag
F,15,52,1,B,I,01,I,GetMItemMark
F,15,54,1,B,I,01,I,GetMItemStyle
F,15,26,1,B,I,00,GetMTitleStart
F,15,30,1,B,I,01,I,GetMTitleWidth
F,15,17,2,B,A,00,GetSysBar
P,15,44,0,B,N,02,I,I,HiliteMenu
P,15,47,0,B,N,00,InitPalette
P,15,13,0,B,N,02,A,I,InsertMenu
P,15,15,0,B,N,03,A,I,I,InsertMItem
F,15,35,1,B,I,00,MenuGlobal
P,15,9,0,B,N,02,A,A,MenuKey
P,15,41,0,B,N,00,MenuNewRes
P,15,11,0,B,N,01,A,MenuRefresh
P,15,43,0,B,N,02,A,A,MenuSelect
F,15,45,2,B,A,01,A,NewMenu
F,15,21,2,B,A,01,A,NewMenuBar
P,15,23,0,B,N,03,I,I,I,SetBarColors
P,15,57,0,B,N,01,A,SetMenuBar
P,15,31,0,B,N,02,I,I,SetMenuFlag
P,15,55,0,B,N,02,I,I,SetMenuID
P,15,33,0,B,N,02,A,I,SetMenuTitle
P,15,36,0,B,N,02,A,I,SetMItem
P,15,40,0,B,N,01,I,SetMItemBlink
P,15,38,0,B,N,02,I,I,SetMItemFlag
P,15,56,0,B,N,02,I,I,SetMItemID
P,15,51,0,B,N,02,I,I,SetMItemMark
P,15,58,0,B,N,02,A,I,SetMItemName
P,15,53,0,B,N,02,I,I,SetMItemStyle
P,15,25,0,B,N,01,I,SetMTitleStart
P,15,29,0,B,N,02,I,I,SetMTitleWidth
P,15,18,0,B,N,01,A,SetSysBar
```

```
H,L,$0100,18,QDAux
P,18,1,0,B,N,00,QDAuxBootInit
P,18,2,0,B,N,00,QDAuxStartUp
P,18,3,0,B,N,00,QDAuxShutDown
F,18,4,1,B,I,00,QDAuxVersion
P,18,5,0,B,N,00,QDAuxReset
F,18,6,1,B,I,00,QDAuxStatus
P,18,9,0,B,N,06,A,A,A,A,I,A,CopyPixels
P,18,10,0,B,N,00,WaitCursor
```

```
H,L,$0100,19,Print
P,19,1,0,B,N,00,PMBootInit
P,19,2,0,B,N,02,I,I,PMStartUp
P,19,3,0,B,N,00,PMShutDown
F,19,4,1,B,I,00,PMVersion
P,19,5,0,B,N,00,PMReset
F,19,6,1,B,I,00,PMStatus
P,19,28,0,B,N,03,A,A,I,LLDBitMap
P,19,27,0,B,N,01,I,LLDControl
P,19,26,0,B,N,01,I,LLDShutDown
P,19,25,0,B,N,02,I,I,LLDStartUp
P,19,29,0,B,N,03,A,I,I,LLDText
F,19,22,1,B,I,00,PrChoosePrinter
F,19,22,1,B,I,00,PrChooser
P,19,15,0,B,N,01,A,PrCloseDoc
P,19,17,0,B,N,01,A,PrClosePage
P,19,9,0,B,N,01,A,PrDefault
F,19,35,1,B,I,00,PrDriverVer
F,19,20,1,B,I,00,PrError
F,19,12,1,B,I,01,A,PrJobDialog
F,19,14,2,B,A,02,A,A,PrOpenDoc
P,19,16,0,B,N,02,A,A,PrOpenPage
P,19,18,0,B,N,03,A,A,A,PrPicFile
P,19,13,0,B,N,03,A,A,I,PrPixelMap
F,19,36,1,B,I,00,PrPortVer
P,19,21,0,B,N,01,I,PrSetError
F,19,11,1,B,I,01,A,PrStlDialog
F,19,10,1,B,I,01,A,PrValidate
```

```
H,L,$0100,22,Scrap
P,22,1,0,B,N,00,ScrapBootInit
P,22,2,0,B,N,00,ScrapStartUp
P,22,3,0,B,N,00,ScrapShutDown
F,22,4,1,B,I,00,ScrapVersion
P,22,5,0,B,N,00,ScrapReset
F,22,6,1,B,I,00,ScrapStatus
P,22,13,0,B,N,02,A,I,GetScrap
F,22,18,1,B,I,00,GetScrapCount
F,22,14,2,B,A,01,I,GetScrapHandle
F,22,16,2,B,A,00,GetScrapPath
F,22,15,2,B,L,01,I,GetScrapSize
F,22,19,1,B,I,00,GetScrapState
P,22,10,0,B,N,00,LoadScrap
P,22,12,0,B,N,03,L,I,A,PutScrap
P,22,17,0,B,N,01,A,SetScrapPath
P,22,9,0,B,N,00,UnloadScrap
P,22,11,0,B,N,00,ZeroScrap
```

```
H,L,$0100,23,StdFile
P,23,1,0,B,N,00,SFBootInit
P,23,2,0,B,N,02,I,I,SFStartUp
P,23,3,0,B,N,00,SFShutDown
F,23,4,1,B,I,00,SFVersion
P,23,5,0,B,N,00,SFReset
F,23,6,1,B,I,00,SFStatus
P,23,13,0,B,N,01,I,SFAllCaps
P,23,9,0,B,N,06,I,I,A,A,A,A,SFGetFile
P,23,11,0,B,N,08,I,I,A,A,A,A,A,A,SFPGetFile
P,23,12,0,B,N,08,I,I,A,A,I,A,A,A,SFPPutFile
P,23,10,0,B,N,06,I,I,A,A,I,A,SFPutFile
```

```
H,L,$0100,25,NoteSyn
P,25,1,0,B,N,00,NSBootInit
P,25,2,0,B,N,02,I,A,NSStartUp
P,25,3,0,B,N,00,NSShutDown
F,25,4,1,B,I,00,NSVersion
P,25,5,0,B,N,00,NSReset
F,25,6,1,B,I,00,NSStatus
P,25,13,0,B,N,00,AllNotesOff
F,25,9,1,B,I,01,I,AllocGen
P,25,10,0,B,N,01,I,DeallocGen
P,25,12,0,B,N,02,I,I,NoteOff
P,25,11,0,B,N,04,I,I,I,A,NoteOn
```

```
H,L,$0100,26,NoteSeq
P,26,1,0,B,N,00,SeqBootInit
P,26,2,0,B,N,04,I,I,I,I,SeqStartUp
P,26,3,0,B,N,00,SeqShutDown
F,26,4,1,B,I,00,SeqVersion
P,26,5,0,B,N,00,SeqReset
F,26,6,1,B,I,00,SeqStatus
P,26,13,0,B,N,00,AllNotesOff
F,26,10,1,B,I,00,ClearIncr
P,26,12,3,B,N,00,GetLoc
F,26,11,1,B,I,00,GetTimer
P,26,10,0,B,N,00,SeqBootInit
P,26,9,0,B,N,01,I,SetIncr
P,26,18,0,B,N,01,A,SetInstTable
P,26,14,0,B,N,03,I,I,I,SetTrkInfo
P,26,19,0,B,N,00,StartInts
P,26,15,0,B,N,02,A,A,StartSeq
P,26,16,0,B,N,00,StepSeq
P,26,20,0,B,N,00,StopInts
P,26,17,0,B,N,01,I,StopSeq
```